Université Claude Bernard Lyon 1

Création d'un Album Photo

DAMIER Alexandre et AUBERT Thierry

Juin 2010

Table des matières

Introduction.....4

1 – Ruby On Rails 5

1.1 – Des notions de bases pour Rails 5

1.1.1 – Le Langage Ruby.....5

1.1.2 – Le Langage HTML 6

1.1.3 – Les feuilles de style CSS 8

1.1.4 – Le JavaScript 10

1.1.4 – La Base de données.....10

1.2 – Le principe du Modèle-Vue-Contrôleur 13

1.2.1 – Le modèle (ActiveRecord).....13

1.2.2 – Le contrôleur (ActionController) 13

1.2.3 – La vue (ActionView) 13

1.3 – Squelette d'une application RoR 15

2 – Notre Application : « PhotosOnRails » 16

2.1 – Son architecture 16

2.1.1 – Navigation pour un utilisateur classique.....16

2.1.2 – Navigation pour l'administrateur.....18

2.1.3 – Les différents éléments du MVC.....19

2.1.4 – La structure de la base de données 22

2.2 – Le fonctionnement sur un exemple.....24

2.3 – Les plugins utilisés.....26

2.3.1 – Description globale des plugins.....26

2.3.2 – Le cryptage des mots de passe par Restful.....27

2.4 – Mise en place.....28

2.4.1 – Les débuts.....28

2.4.2 – Installation de Rails (pour Ubuntu) 29

2.4.3 – Création d'un « mini » projet Rails 29

2.4.4 – Les premiers pas de notre application.....29

2.4.5 – Le rajout des autres fonctions 36
2.5 – Difficultés rencontrées 39
2.5.1 – De nouvelles notions 39
2.5.2 – Des problèmes logiciels 39
2.5.3 – La coordination du travail 40
2.6 – Graphisme du site 41
2.6.1 – Gimp 41
2.6.2 – Vues d'ensemble 43

3 – Audit de l'application 45
3.1 – Types de failles recherchées, solutions et outils appropriés 45
3.1.1 – Stockage des mots de passe 45
3.1.2 – Jeu avec les URL 47
3.1.3 – Transfert des données 47
3.2 – Autocritique de l'application 50
3.2.1 – Ordonnancement du projet 50
3.2.1 – Agencement du MVC et de la BDD 50

Conclusion 51

Annexes 52
Annexes pour « Photos » 52
Annexes pour « Albums » 53
Annexes pour « MenuAlbums » 54
Annexes pour « Coverflows » 55
Annexes pour « Users » 56
Annexes pour « Sessions » 58
Annexes pour l'application 59
Tutoriel Ruby On Rails 60

Introduction

Au départ le sujet de notre TER s'articulait autour d'une problématique simple, l'organisation et la gestion d'un ensemble de photos numériques sous la forme d'un album photo consultable par le biais d'un navigateur web Pour cela il était demandé la réalisation d'un script shell ou d'une fonction dans le langage de programmation désiré qui, à partir d'un dossier de photos, devait recréer l'arborescence et permettre une gestion optimisée des photos. Au regard de ce sujet, notre choix c'est tout d'abord naturellement porté sur le choix du langage Ruby, étant un langage libre orienté objet. Cette décision fut prise pour plusieurs raisons, tout d'abord la découverte d'un nouveau langage de programmation totalement inconnu pour nous, son côté orienté objet le rendant simple de compréhension, ainsi que les quelques échos favorables que nous avons eu par le biais de certains de nos professeurs comme M. Y.Gérard. Mais au regard des contraintes actuelles et de notre environnement quotidien où les interfaces sont de plus en plus prépondérantes, élaborées et fonctionnelles, il nous aurait paru obsolète de présenter cela sous la forme d'un fichier à exécuter par les utilisateurs. D'autant plus que l'essor que connait actuellement le langage Ruby est en grande partie dû au Framework Ruby On Rails, celui-ci étant tout particulièrement adapté pour le développement d'applications web. C'est donc au final pour toutes ces raisons que notre TER c'est naturellement tourné sur la création d'une application web pour la gestion d'images, ce qui nous offre une opportunité supplémentaire d'élargir notre champ de connaissances à d'autres domaines connexes tels que par exemple les bases de données.

Dans un premier temps nous parlerons plus généralement du Framework Ruby On Rails (ou RoR) et des compétences que celui-ci requiert pour la construction d'une application. Nous aborderons d'ailleurs dans cette même partie de manière plus particulière le fondement même de Rails, c'est-à-dire le motif de conception Modèle-Vue-Contrôleur dit MVC.
Dans un deuxième temps nous rentrerons plus dans les détails spécifiques à notre application, en présentant son architecture, un exemple de fonctionnement, les plugins utilisés ainsi que toutes les étapes et difficultés de mise en place que nous avons du effectuer. Nous finirons ensuite cette partie sur l'aspect graphique du site.
Pour finir nous porterons un regard critique sur notre travail en exposant les failles recherchées, notamment en termes de sécurité, ainsi que les solutions s'y rapportant en finissant sur une autocritique de l'application.

1 - Ruby On Rails

1.1 - Des notions de bases pour Ruby On Rails

Comme précédemment précisé dans l'introduction, Ruby On Rails est un framework basé sur Ruby utilisé pour le développement d'applications sur le web. A ce titre différentes notions d'informatiques gravitent autour, nous présenterons dans la suite de cette partie ces différentes notions.

1.1.1 - Le langage Ruby

Ruby est un langage orienté objet. Voici quelques petites indications sur la syntaxe de ce langage, une petite introduction nécessaire à la bonne compréhension du projet à travers un exemple de base.

```
1.  class Essai < AutreClasse
2.  tracer :tout
3.  def exemple
4.  @message = Bonjour.new
5.  amis = Copains.liste
6.  for a in amis
7.  @message.saluer(a)
8.  end
9.  end
10.   end
```

Ligne 1: Définition d'une classe `Essai` qui hérite de la classe `AutreClasse`. L'héritage signifie entre autre que celle-ci dispose des méthodes écrites dans la surclasse.
Ligne 2: Rajout d'une fonction `tracer` (définie autre part) dans cette classe.
Ligne 3: Définition d'une nouvelle méthode exemple.
Ligne 4: On crée un Objet de classe `Bonjour`. @ désigne une variable d'instance, et il n'est donc pas nécessaire de la déclarer avant.
Ligne 5: Appel de la méthode liste sur l'objet `Copains`.
Ligne 6: `amis` et `a` sont des variables locales, il n'est donc pas nécessaire de les déclarer.
Ligne 7: Envoie le message `saluer` à l'objet `@message` (avec le paramètre `a`). Envoyer un message sur un objet ou sur une classe utilise la même syntaxe.
Lignes 8 ,9 et 10 : Fin de la boucle `for`, puis de la méthode `exemple` et enfin de la classe.

1.1.2 - Le langage HTML

L'Hypertext Markup Language, ou HTML, est le format de données conçu pour représenter les pages web. C'est un langage de balisage qui permet d'écrire de l'hypertexte, d'où son nom. HTML permet également de structurer sémantiquement et de mettre en forme le contenu des pages, d'inclure des ressources multimédias dont des images, des formulaires de saisie, et des éléments programmables tels que des applets. Il est souvent utilisé conjointement avec des langages de programmation (JavaScript) et des formats de présentation (feuilles de style css dont nous parlerons plus tard).

a) La Structure des Documents HTML

Un document HTML est séparé entre un en-tête et un corps. L'en-tête contient les informations sur le document comme par exemple son titre. Le corps contient quant à lui ce qui est affiché. Il est également possible d'indiquer la langue de n'importe quelle partie du document et de gérer le mélange de texte s'écrivant de gauche à droite avec du texte s'écrivant de droite à gauche.
Par convention l'extension donnée au fichier est .htm ou .html, mais une page web peut potentiellement porter n'importe quelle extension. En effet dans notre cas les fichiers portent des extensions .erb du fait que nos pages web comportent du code en langage Ruby.
Il est conseillé d'indiquer dans la page HTML une référence à la norme HTML utilisée, afin de spécifier le standard utilisé pour le codage de la page. Cette déclaration se fait par une ligne du type :

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN">
<HTML>
<HEAD>Titre de la page</HEAD>
<BODY>Contenu de la page</BODY>
</HTML>
```

b) Quelques notions utilisées dans notre projet

L'écriture en gras : `ce mot est en <b>gras</b>`
ce mot est en **gras**

L'écriture en italique : `ce mot est en <i>italique</i>`
ce mot est en *italique*

L'écriture en souligné : `ce mot est en <u>souligné</u>`
ce mot est en souligné

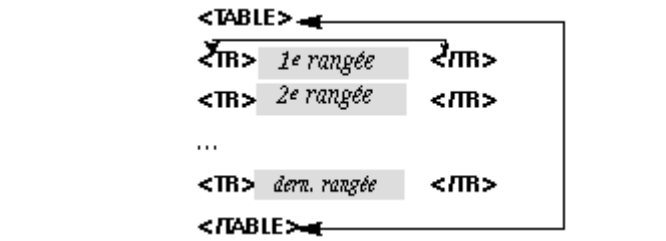
A titre illustratif car nous verrons dans la partie des feuilles de style (CSS) que ces balises sont alors inutiles dans la page HTML

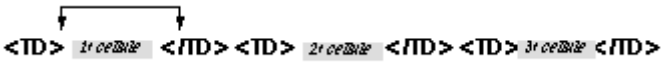
La balise `<br></br>` : Indique un retour à la ligne. Elle n'entraine pas pour autant de saut de ligne. Il est possible d'utiliser la balise `<br>` à l'intérieur d'autres tags (ex: `<p>`, `<td>`, `<label>` ...)

La balise `<p>` de saut de paragraphe : Force un retour suivi d'un espacement vertical plus important.

Les titres `<h1></h1>,<h2></h2>, etc.` : Sont des en-têtes servant à intituler un ensemble de paragraphes. Le texte qui s'y trouve est en général affiché dans une police de taille plus importante que le reste du texte, et en gras, et est séparé du texte qui le précède et qui le suit par un espace plus important.

Les tableaux `<table></table>` : Ils sont composés de cellules, agencées en rangées, et les rangées sont alignées les unes en dessous des autres. Les directives de base pour le construire procèdent donc de cette façon:
On peut déterminer si les réglures seront visibles ou non, en rajoutant un complément d'information dans la directive se rapportant à tout le tableau, par exemple `<table border>` indique qu'elles doivent être affichées. On ferme un tableau avec la syntaxe habituelle `</table>`.
`<tr>` signifie «**T**able **R**ow» (rangée de tableau, ou encore ligne), chaque rangée est elle-même constituée de cellules (colonnes), délimitées par les directives `<td></td>` .





Les formulaires interactifs `<form></form>` : Ils sont utilisés pour introduire des éléments interactifs permettant par exemple un dialogue avec les utilisateurs. L'utilisateur saisit des informations en remplissant des champs puis appuie sur un bouton de soumission (submit). Nous ne rentrerons pas plus dans les détails, les balises de ce type présentent dans notre projet étant générées par du code ruby dont le résultat est interprété sous cette forme.

Les balises `<div></div>` : Elles sont utilisées pour regrouper des éléments HTML entre eux et créer des propriétés de feuilles de style (ou css) à un endroit précis , pour cela nous leur attribuons un nom de la manière suivante : `<div id = name> </div>` permettant de les identifier dans nos fichiers css. Nous détaillerons dans la partie suivante ce que sont ces fameuses feuilles de style css.

Voici donc les rudiments de HTML qui devraient être suffisants à la compréhension des éléments de vues dans notre projet. Bien entendu de nombreux autres éléments existent, et même dans ceux cités, des variantes sont possibles dépendant d'une part du standard utilisé et d'autre part des besoins qui nous sont propres.

1.1.3 - Les feuilles de style CSS (Cascading Style Sheets)

Le langage CSS est utilisé pour définir l'aspect d'un site web, comme par exemple la couleur du fond de la page ou le type de police, l'emplacement des éléments, ... Plus concrètement, une feuille de style est un petit fichier (extension .css) dans lequel sera définit cet aspect.

Les avantages de l'utilisation des feuilles de style :

- La structure et la présentation des pages web sont gérées séparément.
- La conception sans se soucier de la présentation,
- Le code HTML est réduit en taille et en complexité.

Un exemple concret est l'utilisation d'une couleur de fond pour vos pages web. Sans les feuilles de style, il faudrait alors déclarer le corps de votre page de la manière suivante : `<body bg color = « #code_couleur_souhaité »>` pour chacune de vos pages HTML . Tandis que la feuille de style permet de le faire en écrivant dans votre .css : `body { background-color: #code_couleur_souhaité; }`, qui sera pris en compte pour chacune de vos pages. L'intérêt prend alors tous son sens lors d'éventuelles modifications futures de l'apparence du site web, ainsi en ne modifiant qu'une ligne du CSS on modifie autant de pages web que vous en avez.

De manière plus large, le CSS permet de définir les caractéristiques visuelles (emplacement, couleur, …) de n'importe quel élément de votre page web.

a) L'utilisation des feuilles de style

Bien que le CSS puisse être écrit directement dans votre page web, nous choisissons ici de vous montrer la méthode consistant à le mettre dans un fichier de type .css à part pour justement bénéficier de l'avantage du « minimum de modifications, maximum de rendu » comme expliqué dans l'exemple.

Tout d'abord, afin que le fichier CSS soit pris en compte dans une page web, il convient d'inclure dans la page une ligne à cet effet à l'endroit approprié :

```
<head>
<title>titre de ma page</title>
<link rel="stylesheet" type="text/css" href="style.css"/>
</head>
```

Où la partie `href` correspond à l'emplacement du fichier CSS (chemin relatif à l'endroit où se situe la page web, ou dans notre cas l'endroit où l'application se positionne par défaut). On notera également ici les balises `<head></head>` déjà présentées. Il ne reste alors plus qu'à remplir notre fameux fichier CSS.

b) Syntaxe des feuilles de style

La syntaxe du CSS : Elle est très simple : `selecteur { propriété : valeur }`. Chaque sélecteur (body dans l'exemple) peut avoir plusieurs propriétés (séparées par des points virgule) avec des valeurs indépendantes .

```
body {
 background: #eeeeee;
 font-family: Trebuchet MS, Verdana, Arial;
}
```

Vous vous rappelez sans doute que nous avions donné un nom à nos balises div : `<div id = nom></div>`, ce nom peut alors être utilisé ici pour le sélecteur.

```
#nom{
  width: 80%;
  padding: 20px;
  border: 1px solid #666;
  background: #ffffff;
}
```

On notera en revanche la présence du # pour utiliser le CSS pour un id.

Si on traduit les trois mots de « Cascading Style Sheets » : On obtient Feuille de style en cascade. Cela provient du fait que si vous définissez une police de type "Trebuchet MS" sur la balise `<body>`, l'ensemble des autres éléments du site prendra comme police Trebuchet MS, inutile de le redéfinir à chaque élément (de même pour les marges, …). Si néanmoins vous souhaitez par exemple modifier la police d'écriture de vos titres `<h1>` … `<h6>`, il vous suffira de la redéfinir pour ces éléments.

Combiner : Vous pouvez aussi combiner les éléments HTML qui regroupent les mêmes caractéristiques

```
h1, h2, h3, h4, h5, h6 {
  color: #009900;
  font-family: Georgia, sans-serif;
}
```

Commentaires : Les feuilles de style peuvent parfois être longues, et il devient alors plus difficile de s'y retrouver. A ce titre il est conseillé d'utiliser des commentaires de la manière suivante : `/* Commentaires */`

Nous ne rentrerons pas plus dans les détails pour cette partie, en effet le nombre de propriétés est immense, et il n'est souvent utile de n'en connaitre que quelques unes pour le rendu désiré. De plus, bien souvent le nom des propriétés étant relativement explicite, cela ne devrait pas gêner à la lecture et la compréhension de nos feuilles de style.

1.1.4 - Le JavaScript

Javascript est un langage de script orienté objet principalement utilisé dans les pages HTML. A l'opposé des langages serveurs (qui s'exécutent sur le site), Javascript est exécuté sur l'ordinateur de l'internaute par le navigateur lui-même. Ainsi, ce langage permet une interaction avec l'utilisateur en fonction de ses actions (lors du passage de la souris au dessus d'un élément, du redimensionnement de la page...). Le JavaScript est placé dans un fichier de type .js ou directement dans le code de la page HTML entre les balises :
`<script language="Javascript"></script>.`
Néanmoins cette dernière solution n'est pas à conseiller, car elle allonge parfois considérablement le code de la page .

- **L'utilisation des JavaScript**

Si le JavaScript est placé dans un fichier .js , afin que celui ci soit pris en compte dans une page web, il convient d'inclure dans la page une ligne à cet effet à l'endroit approprié de la même manière que pour les feuilles de style:

```
<head>
<title>titre de ma page</title>
<script language="javascript" src="chemin_du_fichier"
type="text/javascript">
</head>
```

En réalité, dans le cas précis de notre application nous utiliserons :

```
<%= javascript_include_tag "nom_du_fichier" %>
```
Pour inclure nos .js

```
<%= stylesheet_link_tag "nom_du_fichier" %>
```
Pour inclure nos .css

Ainsi l'application ira récupérer ces fichiers directement à des endroits spécifiques prévus pour les CSS et JavaScript.

Nous n'expliquerons pas la syntaxe du Javascript dans le cadre de notre TER, en effet ceux-ci n'ont pas été réalisés par nous même et ne sont utilisés que pour obtenir un meilleur rendu visuel de notre application web.

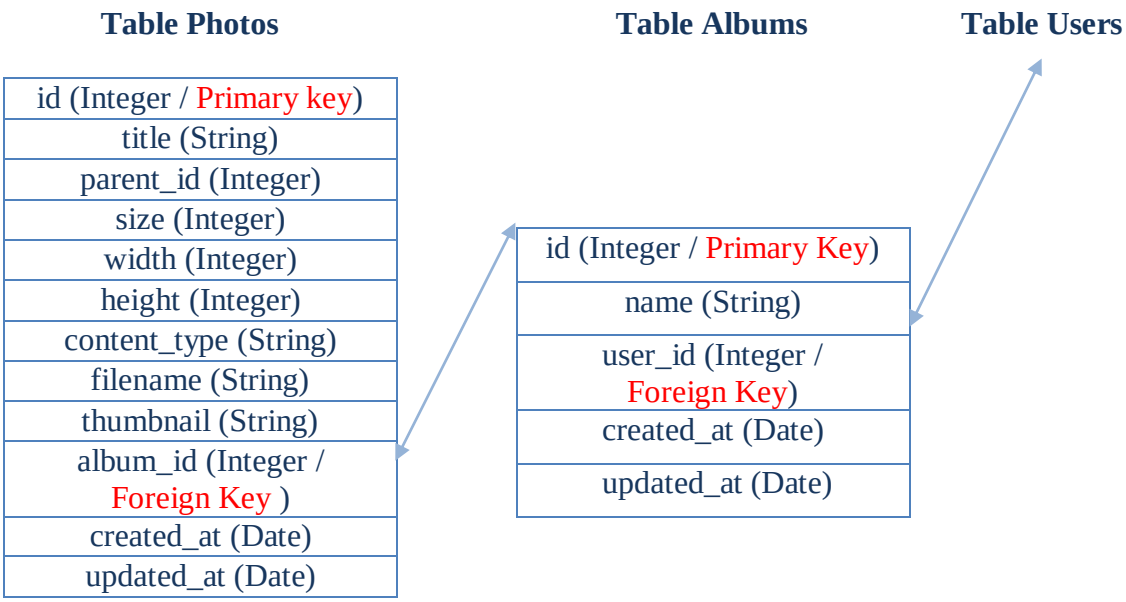
1.1.5 - La Base de Données (ou BDD / DB)

C'est un ensemble structuré d'informations. Une base de données doit être conçue pour permettre une consultation et une modification aisée de son contenu, si possible par plusieurs utilisateurs en même temps. Les données sont stockées dans des champs d'un type déterminé (entier, booléen, date, etc.), et ces champs sont groupés dans des tables, elles même reliées entre elles.

Par défaut Ruby On Rails utilise un système de bases de données (SQLite3), notre choix s'est en revanche porté sur MySQL car il fait partie des logiciels de gestion de base de données les plus utilisés au monde, autant par le grand public que par des professionnels, et il nous a paru préférable de se confronter au système que nous aurions le plus de chance de rencontrer par le futur.

- **Explication par l'exemple**

Dans notre cas l'application possède donc sa base de données (en réalité trois, une pour le développement, une pour les tests, et une pour la mise en production, mais cela n'est pas important ici). Notre base de données possède de nombreuses tables, parmi lesquelles les tables Albums et Photos. Enfin, dans ces tables se trouvent les champs, qui définissent respectivement les caractéristiques de nos albums et de nos photos. Voici un schéma explicatif :



Les différents types de champs :
- Champs numériques (entier, décimal, réel) : informations calculables
- Champs textes : données alpha-numériques
- Champs date et heure
- Champs booléens (oui/non)

Explications des champs les plus importants:
- Le champ id : Entier utilisé comme clé primaire. Une clé primaire est une contrainte d'unicité qui permet d'identifier de manière unique un enregistrement dans une table. Ce champ est auto-incrémenté, c'est à dire qu'à chaque enregistrement dans cette table, l'id augmente de 1. Cela assure que chaque enregistrement à un id unique le caractérisant.

- Le champ album_id de la table Photos : Entier utilisé comme clé étrangère. Une clé étrangère est une contrainte qui garantit l'intégrité référentielle entre deux tables. En d'autres termes cela permet de lier ces deux tables, il s'agit d'un des principes fondamentaux des bases de données relationnelles telles que MySQL.
Ainsi on ne peut pas insérer un enregistrement dans la table Photos avec un id d'Albums qui n'existe pas dans la table Albums et on ne peut pas supprimer un enregistrement de la table Albums si au moins un enregistrement de la table Photos a une valeur d'id d'Albums correspondant à l'enregistrement à supprimer.
- Le Champ user_id de la table Albums sert de la même manière de clé étrangère et permet de lier les Tables Albums et Users (non décrite ici).
- Les autres champs servent à stocker des propriétés propres aux enregistrements, comme la date de création d'un album ou d'une photo, sa taille, son nom …

Dans le cadre de Ruby On Rails il n'est pas nécessaire de construire les bases de données « à la main » en passant par les commandes de MySQL, en effet celui-ci fournit des commandes propres à RoR (rake db :create et rake db:migrate) permettant de créer les bases de données et d'exporter un modèle de tables que l'on définit dans des fichiers Ruby au sein de l'application.

1.2 - Le principe du Modèle-Vue-Contrôleur

Ce principe est un des fondements même de RoR, toute l'architecture d'une application crée via ce Framework en depend.

1.2.1 - Le modèle (ActiveRecord)

ActiveRecord est un "ORM" (Object/Relational Mapper), il assure le lien entre le monde objet de Ruby et le monde relationnel de la base de données. Cela couvre à la fois la structure (comment représenter une table en une classe) et le comportement, autrement dit les problèmes comme la validation des données, l'optimisation des requêtes, ...

Rails interroge la base de données pour en découvrir la structure

Le développeur respecte quelques conventions de nommage

Des indications peuvent être ajoutées Pour guider Rails (validations, associations ...)

1.2.2 - Le contrôleur (ActionController)

Dans les contrôleurs se situent en quelque sorte la logique de l'application. Autrement dit, préparer des données pour une page, et le faire différemment pour un utilisateur authentifié ou anonyme, recevoir les résultats d'un formulaire HTML pour enregistrer des données, et immédiatement rediriger vers une autre page, vérifier qu'un utilisateur est identifié et si non le rediriger vers la page adéquate…

L'enchaînement d'un contrôleur à l'autre se fait suite à des actions de l'utilisateur, ou selon la logique d'un contrôleur, les contrôleurs possèdent également des filtres. Tout cela peut paraitre très vague, et sera illustré également plus tard dans ce rapport.

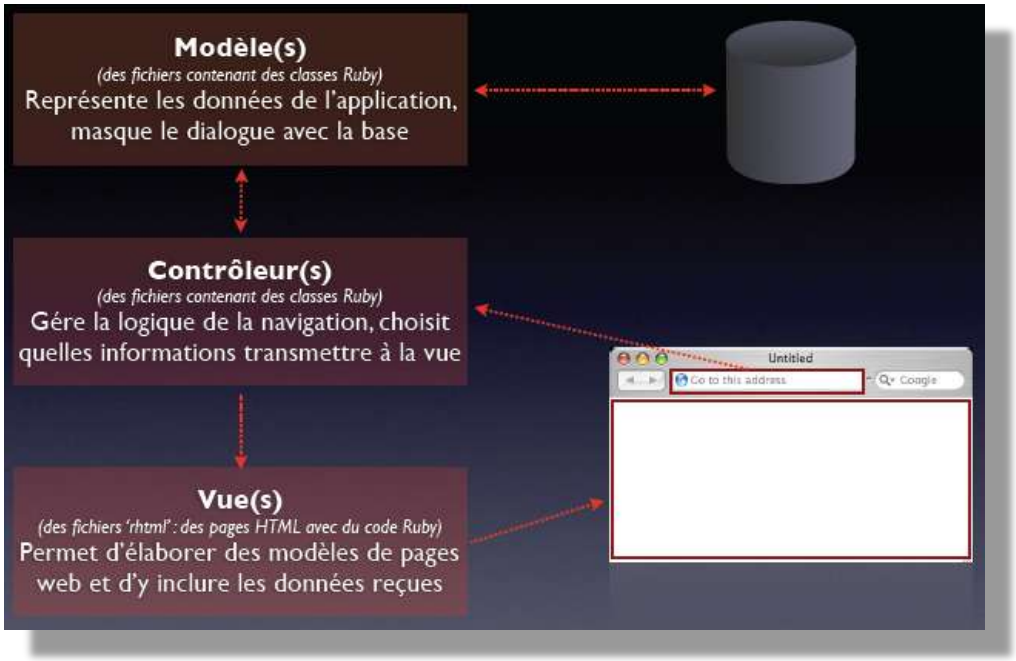
1.2.3 - La vue (ActionView)

Elle utilise le principe de "gabarits" de pages HTML comme présenté précédemment avec un peu de code inclus (fichiers .erb). Les données du contrôleur sont automatiquement transmises à la vue et les liens entre les pages, les formulaires s'expriment en tant que contrôleurs et actions

- **Inclure du code ?**

<% ... %> : Le code Ruby à l'intérieur de ce tag sera exécuté, mais le résultat ne sera pas placé dans la page HTML (les tests, les boucles, …)
<%= ... %> : Le code Ruby à l'intérieur de ce tag sera exécuté, et le résultat sera inclus dans la page HTML (Pour afficher les informations transmises par le contrôleur)

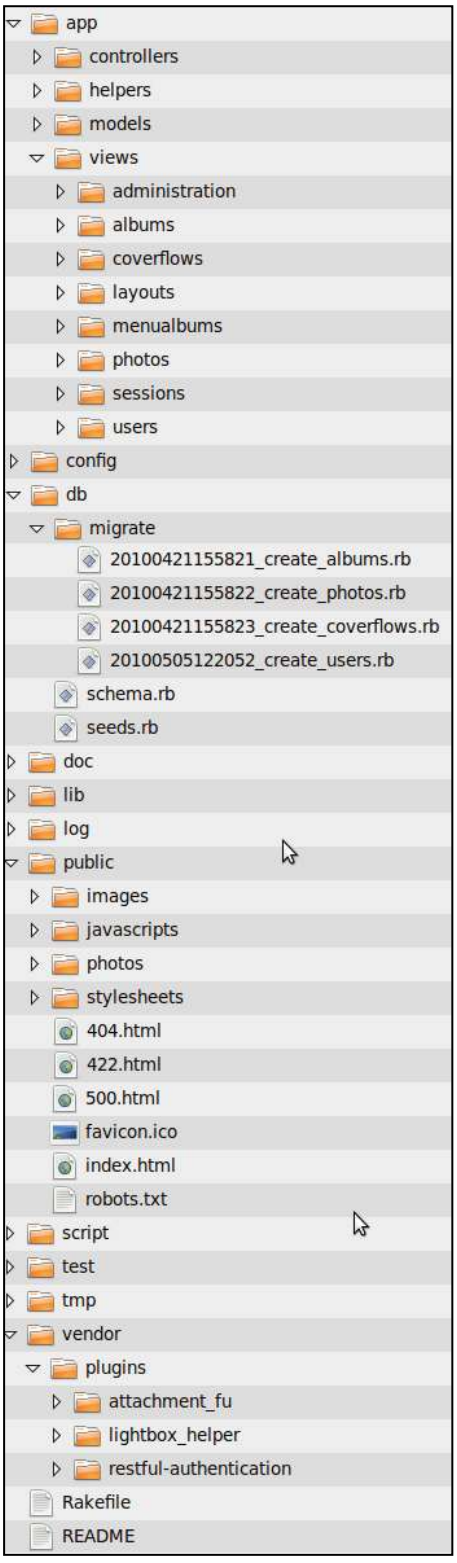
En résumé : « Modèles + Contrôleurs + Vues = Ruby On Rails »



Lorsqu'un client envoie une requête à l'application :

- la requête envoyée depuis la vue est analysée par le contrôleur,
- le contrôleur demande au modèle approprié d'effectuer les traitements,
- le contrôleur renvoie la vue adaptée, si le modèle ne l'a pas déjà fait.

1.3 - Squelette d'une application RoR



Rails est en cela utile qu'il fournit au développeur une structure pour son application, des fichiers, dossiers et sous dossiers qui peuvent dérouter au début mais qui s'avèrent au final très utiles car ils imposent une certaine rigueur et logique tout au long du développement.

Voici donc pour illustrer le squelette de notre projet Ruby On Rails. Nous allons brièvement décrire les principaux emplacements.

Le dossier app : Il contient les sous dossiers controllers (où se situent nos contrôleurs), models (où se situent nos modèles), views (où se situent dans divers dossiers les fichiers HTML correspondant à nos vues) et helpers (les fonctions présentent à l'intérieur servent à rendre la création de la vue plus facile, nous ne les avons pas utilisés de notre côté)

Le dossier config : Contient plusieurs fichiers importants, comme la feuille de routes (que nous n'aborderons pas dans cette partie), le fichier database.yml permettant de renseigner les informations nécessaires pour que l'application puisse dialoguer avec MySQL, etc.

Le dossier db : Contient les fichiers nécessaires à la création et migration des tables de la base de données. Par exemple le fichier schema.rb écrit donc en Ruby donne une description des tables et des champs.

Le dossier public : Comme son nom l'indique, il ne contient aucune donnée sensible. Y sont stockées par exemple les feuilles de style (Dans stylesheets), les JavaScript (Dans javascripts), les photos des utilisateurs (Dans photos) ainsi que les images de l'application (Dans images). Nous remarquons aussi la page d'index de l'application, ainsi que les différentes pages d'erreurs (404, 422 et 500)

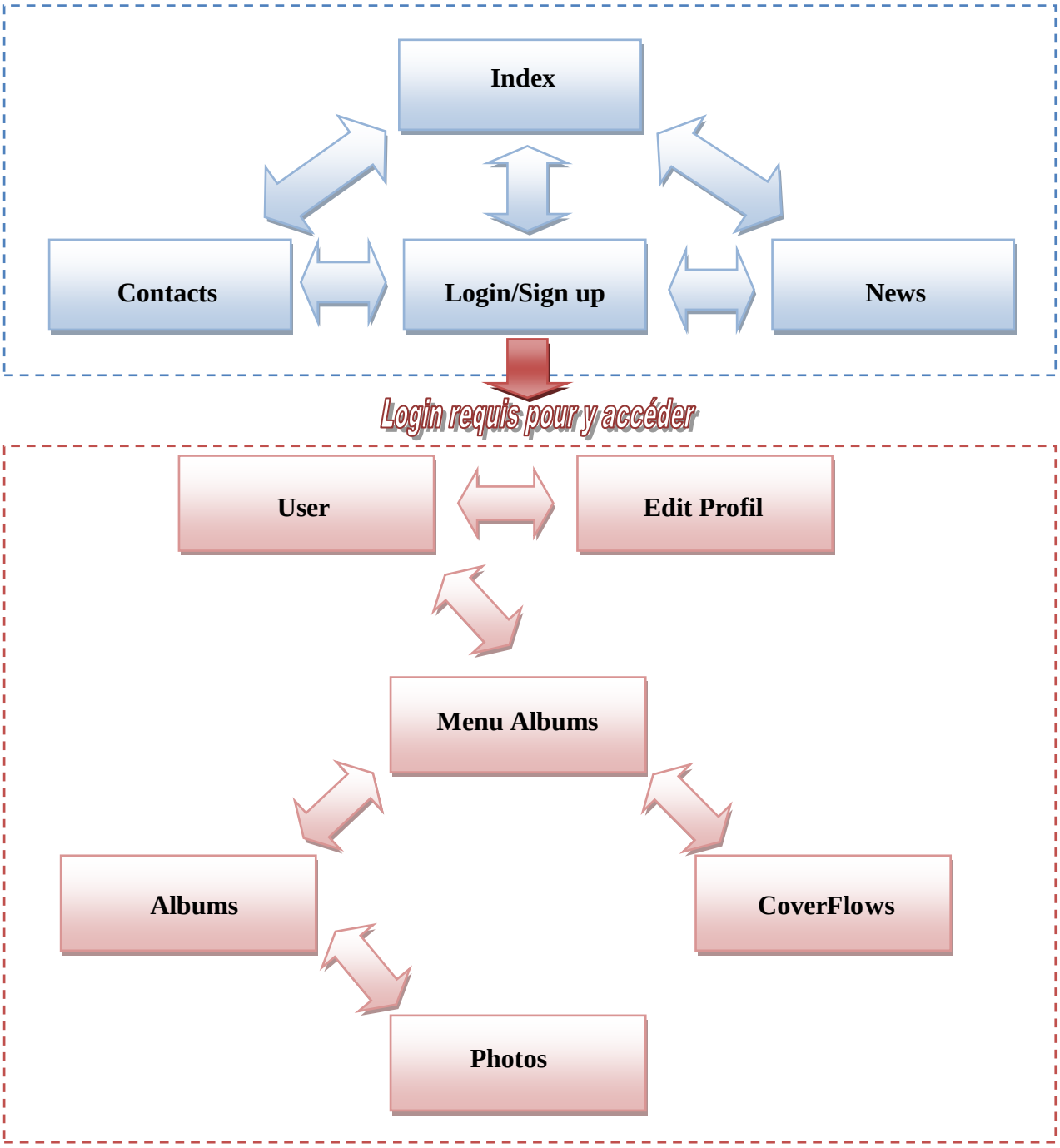
2 -Notre Application : « PhotosOnRails »

2.1- Son architecture

2.1.1 - Navigation pour un utilisateur classique

a) Schéma de navigation

Voici le schéma représentant la navigation générale d'un simple utilisateur dans notre application :



b) Explications pour le schéma de navigation

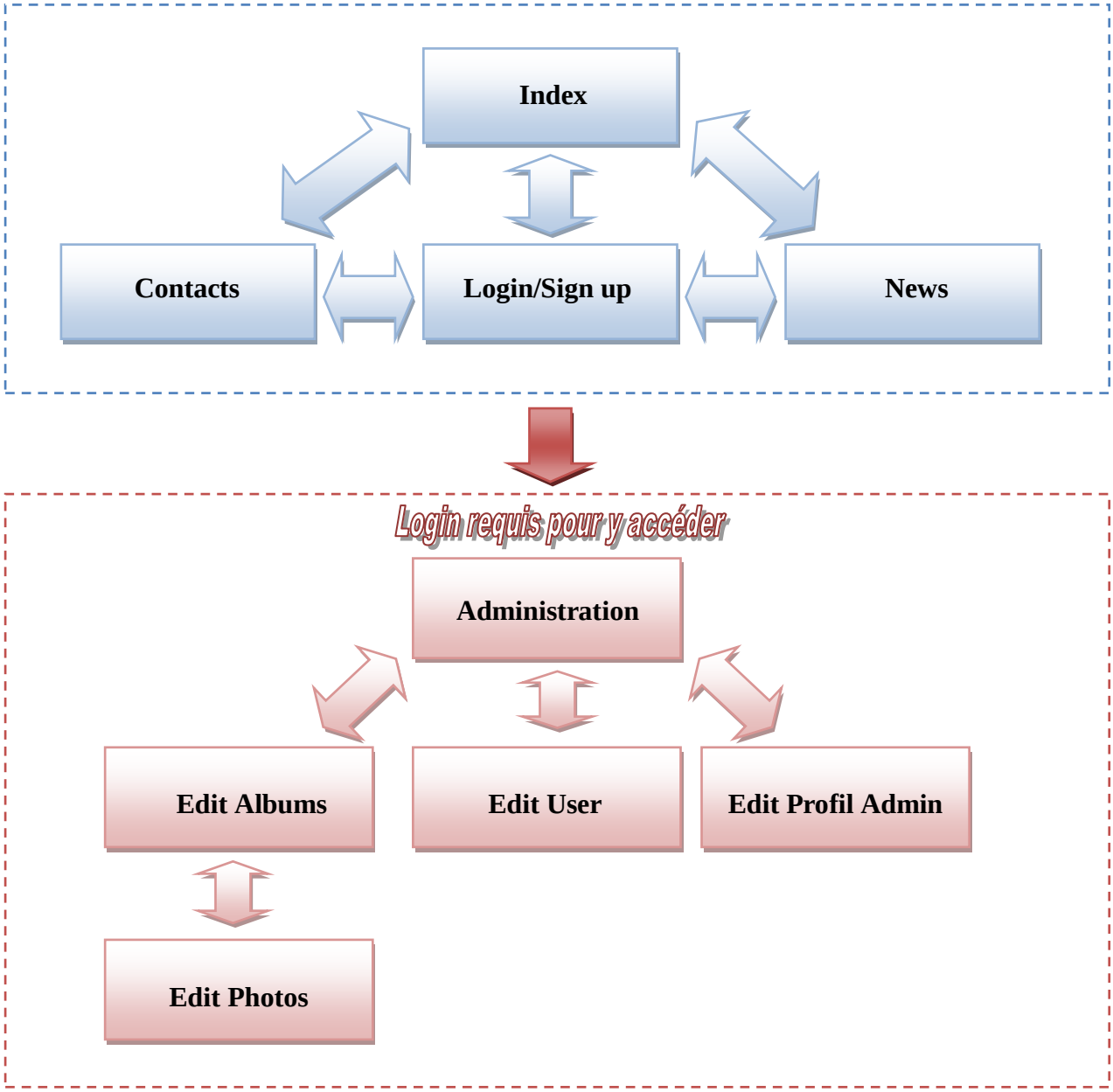
La zone publique : Ici représentée en bleue, elle est accessible à la fois aux visiteurs et aux utilisateurs logués. Chacune de ses composantes est notamment accessible via des liens dans le layout, autrement dit via la zone du site qui demeure identique quelque soit l'endroit où l'on se situe.
On y retrouve notamment l'index du site, une page pour obtenir les coordonnées des administrateurs/créateurs du site, une zone « news » qui tient au courant des avancées et nouveautées mises en place, ainsi que la zone pour se loguer ou s'enregistrer sur le site.

La zone privée : Ici représentée en rouge, elle nécessite obligatoirement d'être un utilisateur logué pour pouvoir y accéder, faute de quoi on se retrouve redirigé sur la page de login. Une fois l'utilisateur logué, il se retrouve redirigé sur l'index des utilisateurs, celui-ci est relativement sommaire et ne comporte qu'un message de bienvenue personnalisé ainsi qu'un lien pour le rediriger sur la page listant tous les albums lui appartenant (Menu Albums). De plus, une fois logué, une zone du layout se modifie et l'utilisateur peut alors cliquer sur un lien afin de modifier ses informations personnelles ou effacer son compte.
Du Menu des Albums il peut accéder à une vision sous forme de coverflow de ses albums et un menu d'édition (De là, un bouton retour est toujours présent).
Le menu d'édition lui permet de changer le nom des albums existants. Il peut également éditer/rajouter des photos d'un album existant ou simplement créer un nouvel album via la partie Photos.

Nous montrerons en image les éléments de navigation dans une autre partie qui sera elle dédiée aux aspects graphiques de l'application.

2.1.2 - Navigation pour l'administrateur

a) Schéma de navigation



b) Explications pour le schéma de navigation

La zone publique : En réalité, l'administrateur est en quelque sorte un utilisateur particulier, à ce titre, il dispose de la même partie publique qu'un utilisateur classique. Il doit donc également passer par la même interface de login pour accéder au panneau d'administration.

La zone privée : Une fois authentifié en tant qu'administrateur, celui-ci est redirigé sur le panneau d'administration listant tous les utilisateurs avec certains renseignements relatifs à eux. Il peut également comme tout utilisateur accéder à l'édition de son propre profil via le lien dans le layout.
Plusieurs liens sont également disponibles devant chaque utilisateur dans la liste. Un lui permet d'accéder à l'édition des Albums et des Photos de l'utilisateur un peu à la manière que l'utilisateur lui-même pourrait le faire. Les deux autres permettent de gérer les utilisateurs, l'un en supprimant le compte, et l'autre en éditant les informations relatifs au profil (login, mot de passe, adresse mail).

Bien qu'un admin soit en réalité un utilisateur avec une fonction particulière au regard de l'application, il a été fait le choix (pour des raisons pratiques et afin de bien distinguer les fonctions) de ne pas permettre à l'administrateur de naviguer tel un utilisateur classique. Autrement dit un admin ne s'occupe que de l'administration à proprement parlé et ne peux créer des albums pour y insérer des photos.

2.1.3 – Les différents éléments du MVC

a) Users

- Un contrôleur
- Un modèle
- Une vue pour s'enregistrer en tant qu'utilisateur
- Une vue pour l'index de l'utilisateur
- Une vue pour éditer son profil

b) Sessions

- Un contrôleur
- Une vue permettant de se loguer en tant qu'utilisateur

c) Administration

- Un contrôleur
- Une vue pour l'index de l'administration
- Les vues de Albums modifiées pour ses besoins

Remarque : Pas de modèle, car il utilise en réalité celui de Users.

d) MenuAlbums

- Un contrôleur
- Un modèle
- Une vue d'index

e) Coverflows

- Un contrôleur
- Un modèle
- Une vue d'index

f) Albums

- Un contrôleur
- Un modèle
- Une vue d'index permettant de détruire l'album et accéder aux autre vues
- Une vue pour changer le nom de l'Album
- Une vue pour changer les Photos de l'album

g) Photos

- Un contrôleur
- Un modèle

Remarque : Pas de vues pour Photos, car leur édition et visualisation est déjà présente dans les autres vues. Tous les modèles et contrôleurs se trouvent en annexe du rapport. A noter également la présence d'un contrôleur nommé Application_controller.rb. Rappelez-vous alors de la syntaxe Ruby pour la définition d'une classe, dans notre cas tous les contrôleurs précités héritent de cette classe.

f) Le routage

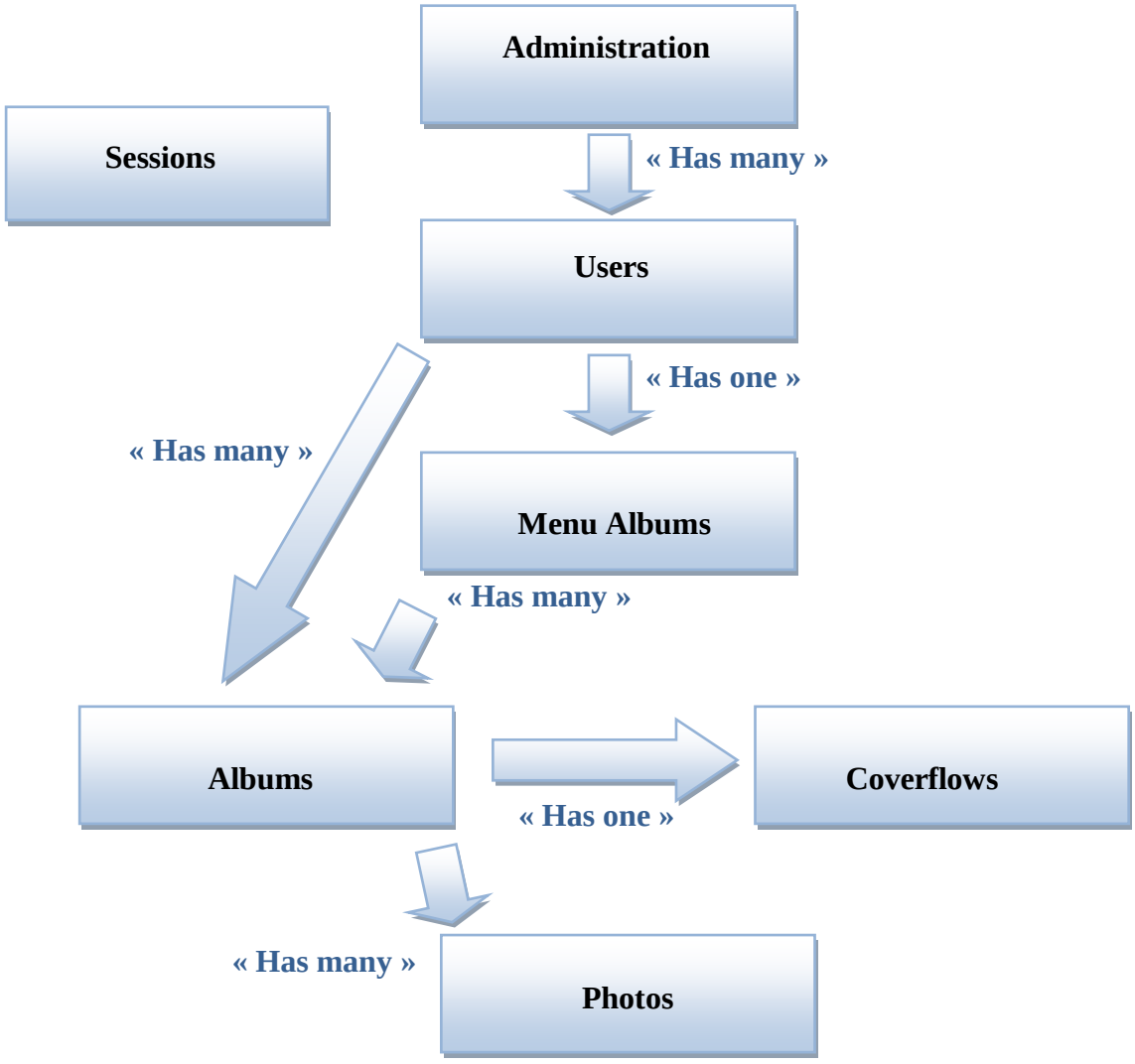
Le système de routage dans Rails est le système qui examine l'URL d'une requête entrante et détermine quelles actions doivent être prises par l'application. Le déclenchement d'une action du contrôleur est le principal événement dans le cycle de vie d'une connexion à une application Rails et il est donc logique que le processus par lequel Rails détermine le contrôleur et l'action à exécuter soit très important. Ce processus est justement inscrit dans le système de routage…

Le routage sert donc à savoir comment interpréter (reconnaitre) une requête URL et comment écrire (générer) une URL. Il permet de faire cela en se basant sur des règles que nous fournissons dans le fichier routes.rb que nous avions localisé précédemment dans le dossier config de notre application. Ces règles sont écrites en utilisant une syntaxe particulière, en fait du code Ruby utilisant des méthodes et paramètres particuliers.

Exemple de notre fichier de routage

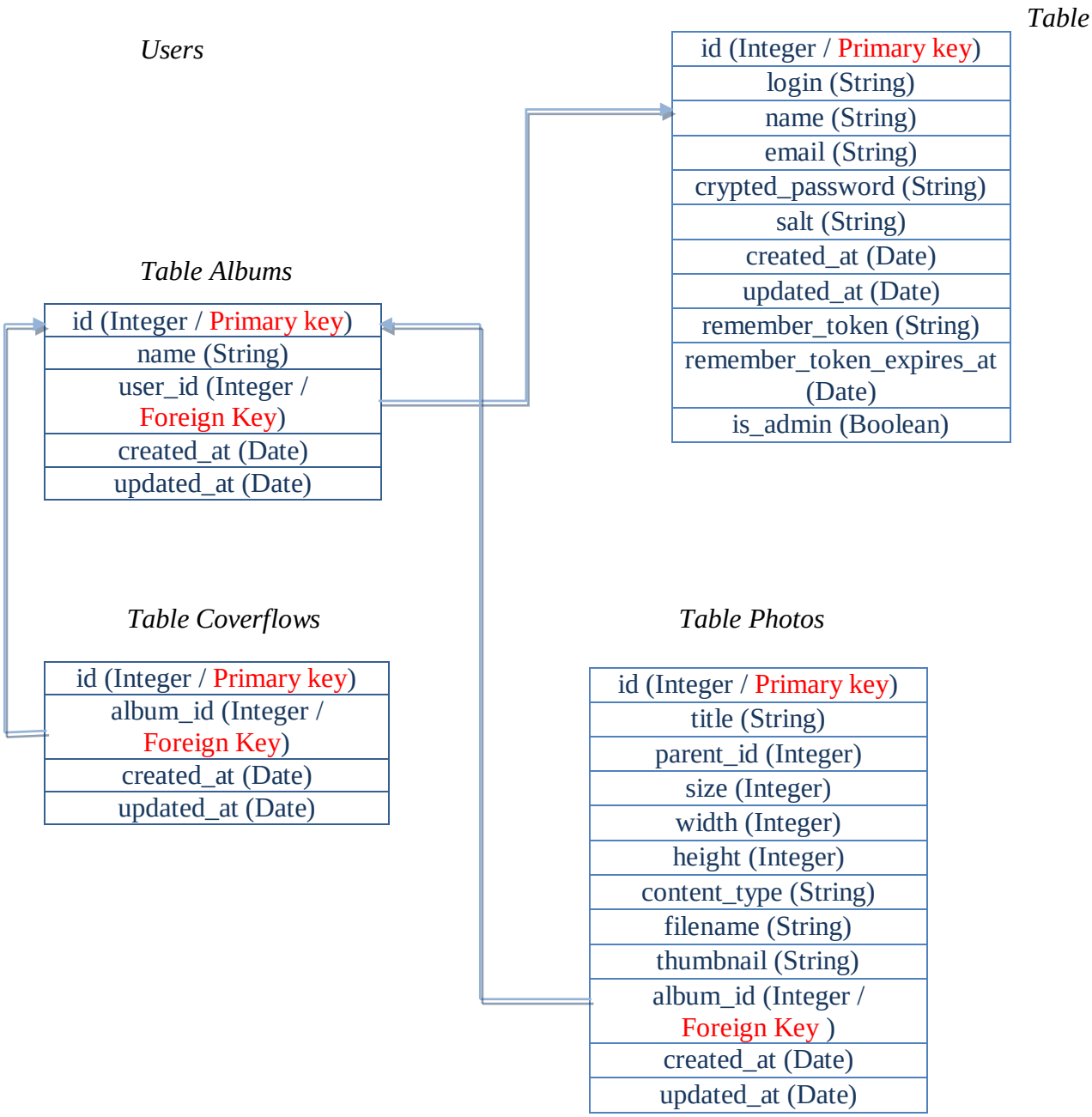
```
ActionController::Routing::Routes.draw do |map|
map.logout '/logout', :controller => 'sessions', :action => 'destroy'
map.login '/login', :controller => 'sessions', :action => 'new'
map.register '/register', :controller => 'users', :action => 'create'
map.signup '/signup', :controller => 'users', :action => 'new'
map.resources :users, :has_many => :albums, :has_one => :menualbum

map.resource :session

map.resources :albums, :has_many => :photos, :has_one => :coverflow
map.resources :menualbums, :has_many => :albums
map.resources :administration, :has_many => :users
map.resources :coverflows, :has_many => :photos

map.connect '*path', :controller => 'application', :action =>'rescue_404'
end
```

Illustration du routage (ressources)



2.1.4 – La structure de la base de données

La manière la plus claire de représenter la base de données dans son intégralité étant un schéma, voici donc la structure de notre BDD :



Certains champs des tables ne sont en réalité pas utilisés dans notre application, cela provient du fait que tous auraient une éventuelle fonction mais que le rendu final de notre application n'a pas pu les intégrer par faute de temps. Nous verrons également plus tard que celle-ci aurait pu être agencée autrement. En réalité la base de données reflète la vision que l'on a d'une application, et il existe autant de variantes que de point de vue, le but étant de trouver la plus efficace en termes de temps et de ressources.

2.2 - Le fonctionnement sur un exemple

Nous allons désormais illustrer un ensemble de notions vues précédemment avec un exemple extrait de notre application.

Contexte : Un utilisateur est enregistré et se trouve sur son « Menualbums » qui liste tous ses albums (nous supposons qu'il en a déjà crée antérieurement). Voyons ce qu'il se passe s'il souhaite visionner un de ses albums en cliquant sur le lien coverflow.

Au départ : `http://PhotoOnRails.com/menualbums`
Contrôleur : Menualbums
Action : Index (car rien n'est précisé derrière dans l'URL)

Clique de l'utilisateur sur le lien « Coverflow »

Lien dans le fichier HTML : `<%= link_to 'Coverflow', :controller =>"coverflows", :action => 'show' , :id => album.id %>`
Le lien fait donc appel au contrôleur Coverflows, plus précisément à son action show, en prenant en paramètre l'id de l'album que l'utilisateur souhaite visionner.

Remarque : Rappelez-vous que la feuille de route nous disait qu'un album « has one » coverlow. Autrement dit à chaque album correspond un coverflow .

Contrôleur Coverflows :

```
class CoverflowsController < ApplicationController
before_filter :login_required
  def show
    if current_user.is_admin
      redirect_to ('/administration')
    end
    @album = Album.find(params[:id])
  End
```

Notre Contrôleur à la ligne 2 comporte le filtre « login_required », ce qui signifie que l'on ne peut faire appel à ce contrôleur que si l'utilisateur est logué, ce qui est supposé être le cas ici.

Regardons plus attentivement la définition de la méthode show. Celle-ci applique un traitement particulier selon que l'on soit un administrateur ou non (autrement dit un simple utilisateur). Notre utilisateur ci-présent n'étant pas admin le traitement à la ligne 7 (`@album = Album.find(params[:id]))` est effectué.

La variable d'instance `@album` est initialisée avec `Album.find (params[:id])` qui va rechercher dans la base de données l'id de l'album en question.

Modèle Coverflow : (au singulier dû aux conventions de nommage de RoR) Il entre alors en jeu. En effet, souvenez-vous, dans le modèle MVC le modèle sert en quelque sorte d'intermédiaire entre le contrôleur et la BDD. Le modèle interroge donc la BDD avec les informations qui lui sont données.

```
class Coverflow < ActiveRecord::Base
  has_many :photos
  belongs_to :album
end
end
```

Pour le modèle un Coverflow est bien relié à un album (belongs_to) et possède effectivement plusieurs photos (has_many) par l'intermédiaire de l'album comme vu sur le schéma de notre BDD.

Sachant donc que nous disposons de l'id de l'album que nous souhaitons visionner et que dans la BDD les photos sont elles même reliées à un unique album, le modèle accède donc aux photos de cet album et retransmet toutes les informations nécessaires au contrôleur. Le contrôleur transmet ensuite à la vue (show.html.erb)

Vue show.html.erb :

Voici la partie importante de la vue

```
<% @album.photos.each do |photo| %>
<img src= "<%=photo.public_filename(:cover)%>"  … />
<% end %>
```

La vue comporte donc une boucle qui va afficher chaque photo récupérée dans la base de données. Dans notre application, le rendu final est modifié par le JavaScript imageflow.js

Au final : http://PhotoOnRails.com/coverflows/112

Contrôleur id de l'album

Remarque : Dans notre exemple la méthode utilisée dans le contrôleur n'apparait pas dans l'url finale, cela est parfois le cas en fonction du type de méthode appelée.

2.3 - Les plugins utilisés

En informatique, un plugin est un logiciel qui complète un logiciel hôte (ici Ruby On Rails) pour lui apporter de nouvelles fonctionnalités. Ils ne peuvent fonctionner seuls car ils sont uniquement destinés à apporter une fonctionnalité à RoR.

2.3.1 - Description globale des plugins

a) Restful Authentication

Le plugin restful authentication est parmi les moyens les plus faciles pour obtenir l'authentification d'un utilisateur dans une application Rails. Il comprend les caractéristiques suivantes : connexion et déconnexion, mot de passe sécurisé (dans la BDD), activation du compte par validation email (non valide à l'état d'avancement actuel de notre application), approbation et la désactivation du compte par email (même remarque), contrôle rudimentaire des autorisations et des accès.

b) Attachment fu

Attachment_fu est un plugin qui permet de faciliter la gestion des fichiers téléchargés vers notre application via des formulaires. Il permet ainsi à chaque utilisateur de rajouter ses photos dans ses albums.
Pour ce faire, attachment fu propose 3 méthodes de stockage:

Le système de fichiers: Les fichiers sont simplement déposés dans un dossier de votre serveur, idéalement public afin qu'ils puissent ensuite être affichés sur votre site (il s'agit de la m »thode utilisée dans notre cas)

Amazon S3: la plateforme de stockage distribuée (et payante) proposée par Amazon.

La base de données: Déjà utilisée pour nos autres données, mais il apparait relativement lourd de stocker des images dans une BDD plutôt que simplement leurs attributs.

c) LightBox

LightBox est un plugin qui permet de créer un effet visuel de zoom sur les images de manière dégradé et agrémenté de boutons pour naviguer d'une image à une autre. Il est basé sur un script très connu : Scriptaculous (ou Script.Aculo.Us) qui est une librairie JavaScript, c'est-à-dire un ensemble de fonctions, méthodes et propriétés JavaScript dont l'objectif est de simplifier les développements dans ce langage. Ses fonctionnalités sont très étendues mais ne nous concernent pas ici.

2.3.2 – Le cryptage des mots de passe par Restful

Pour crypter les mots de passe dans la BDD Restful utilise une méthode hachage avec clé salt. Le hachage est le résultat d'un algorithme qui analyse une chaine de caractère pour la rendre incompréhensible à la lecture. Un algorithme de hachage génère une valeur unique à partir de l'entrée envoyée. L'avantage du hachage est qu'une fois encodé, il est pratiquement impossible de retourner en arrière pour connaître la valeur de départ. Autrement dit, c'est un encryptage unidirectionnel.

Le hachage est fréquemment utilisé pour l'encryptage de mot de passe dans la base de données comme dans notre cas. En utilisant ce procédé, il est impossible pour un visiteur ou un administrateur de la BD de connaître les mots de passe des utilisateurs.
Il est également rajouté une chaîne unique à chaque encryptage appelé *salt*. Cette chaîne permettra de ne pas avoir deux fois le même mot de passe dans la BDD. Cela augmente donc la protection. Cette chaîne est un élément propre à chaque utilisateur, comme cela apparait sur le schéma représentant la BDD.

2.4 - Mise en place

2.4.1 - Les débuts

Au tout départ de la création de notre application, nous ne savions pas vraiment par où commencer. Aussi nous avons débuté par nous familiariser avec quelques notions de langage Ruby via des tutoriels divers.

Comme par exemple : *http://ruby-doc.org/docs/beginner-fr/xhtml/ch01.html*

Assez rapidement, et après avoir retenu les quelques bases Ruby, notre travail s'est guidé vers l'apparence de site que nous devrions construire sous l'impulsion de notre professeur référent. En effet, avoir une certaine idée de l'agencement du site se devait de passer par la création de quelques pages web pour éclaircir un peu plus nos idées qui étaient à ce stade encore très vagues. C'est donc en testant d'écrire quelques pages web que nous nous sommes familiarisés avec la syntaxe du HTML, notamment les balises `<div>` et l'intégration de JavaScript comme ImageFlow (Pour un effet de coverflow) dans ces mêmes pages de la manière suivante :

Dans le header des pages HTML devant utiliser cet effet visuel :

```
<link rel="stylesheet" href="imageflow.css" type="text/css" />
<script src="imageflow.js" type="text/javascript"></script>
```

Puis dans le body des pages HTML :

```
<div id="unique_name" class="imageflow">
        <img src="dir/image_1.jpg" longdesc="URL_1"
        width="w_1" height="h_1" alt="Text_1" />
        <img src="dir/image_2.jpg" longdesc="URL_2"
        width="w_2" height="h_2" alt="Text_2" />
        <img src="dir/image_3.jpg" longdesc="URL_3"
        width="w_3" height="h_3" alt="Text_3" />
    </div>
```

En prenant soin de retenir la partie « unique_name » afin de le reporter dans le fichier .js du JavaScript au niveau de ces quelques lignes :

```
domReady(function()
{
var instanceOne = new ImageFlow();
instanceOne.init({ ImageFlowID: 'unique_name' });
});
```

Ces lignes créent en fait une instance d'ImageFlow une fois que nos pages ont fini de charger leur structure.

2.4.2 - Installation de Rails (pour Ubuntu)

Installation de Ruby: `sudo apt-get install ruby` (sudo servant à lancer cette commande en tant que root sous linux)
Installation de RubyGems: Sur *http://rubyforge.org/frs/?group_id=126*, téléchargement du package désiré, puis dans ce qui a été récupéré se situe un fichier install.rb (autrement dit un fichier Ruby), nous allons donc utiliser Ruby pour l'exécuter : `sudo ruby install.rb`
Une Gemme est une application ou une bibliothèque rassemblée pour obtenir un élément d'installation unique.
Installation de Rails : Il suffit alors de lancer la commande `sudo gem install rails --include-dependencies`

2.4.3 - Création d'un « mini » projet Rails

Pour nous familiariser véritablement avec RoR et ne pas se lancer trop rapidement dans notre application au risque d'être perdu, nous avons commencé par la création d'une application basique servant à créer des fiches de films. Pour cela deux tutoriels ont été suivis (en annexe).

2.4.4 - Les premiers pas de notre application

Ces premières commandes furent en parallèle écrites également dans un script shell afin de pouvoir aisément reproduire ces actions en cas d'erreurs futures de notre part. Cette méthode fut rapidement abandonnée car l'intérêt était faible au vu de l'implication que cela nous aurait demandé de se familiariser plus amplement avec l'écriture de scripts shell.

Remarque : Dans les captures d'écran qui suivent, celles-ci correspondant au projet final de notre application certains éléments peuvent apparaitre alors que leur écriture a été faite ultérieurement.

Generation du squelette de l'application :
rails PhotosOnRails

Generation des modèles et contrôleurs de Photos et Albums :
Cd PhotosOnRails (Pour se placer à la racine de notre application)
script/generate model photo name :string
script/generate model album name :string
script/generate controller albums
script/generate controller photos

Installation du plugin attachment fu :
Script/plugin install http://svn.techno-weenie.net/projects/plugins/attachment_fu

Installation du plugin Lightbox :
Script/plugin install http://lipsiasoft.googlecode.com/svn/trunk/lightbox_helper

Inclusion des JavaScript et Stylesheets des plugins dans le layout (vue de l'application) comme vu précédemment (i.e. Entre les balises <head></head>) afin que ceux-ci soient pris en compte à n'importe quel endroit de l'application.

```
<%= stylesheet_link_tag 'style' %>
<%= stylesheet_link_tag 'lightbox' %>
<%= javascript_include_tag :defaults %>
<%= javascript_include_tag "scriptaculous" %>
```

Ecriture des contrôleurs Albums et Photos :

Les éléments concernant les redirections selon si l'utilisateur est logué ou non, et admin ou non ont été rajoutés dans des étapes ultérieures. Voici l'allure des fichiers correspondant. Ceux-ci ne seront par la suite pas tous inclus dans le rapport mais en annexe.

```
class PhotosController < ApplicationController
before_filter :login_required
  def create
    @album = Album.find(params[:album_id])
    @photo = Photo.new(params[:photo])

    if @photo.save
      @album.photos << @photo
      redirect_to album_url(@album)
    else
      flash[:notice] = "5mo max"
      redirect_to album_url(@album)
    end

  end

  def destroy
    _find_album
    @photo = @album.photos.find(params[:id])
    @photo.destroy
    redirect_to album_url(@album)
  end

  def _find_album
    @album = Album.find(params[:album_id])
  end


end
```

Fonction de création des photos

Fonction de destruction

Fonction pour rechercher les photos relatives à un album donné

```ruby
class AlbumsController < ApplicationController
before_filter :login_required

  def index
    @albums = Album.find :all
    if current_user.is_admin
      redirect_to ('/administration')
    end
  end

  def new
    @album = Album.new
    if current_user.is_admin
      redirect_to ('/administration')
    end
  end

  def show
    @album = Album.find(params[:id])
      @photo = Photo.new
    else
  end

  def edit
    @album = Album.find(params[:id])
  end

  def create
    if current_user.is_admin
      redirect_to ('/administration')
    end
    @album = Album.new(params[:album])
    @user=current_user
    @user.albums << @album
    if @album.save
      redirect_to albums_url
    else
      render :action => "new"
    end
  end

  def update
    @album = Album.find(params[:id])

    if @album.update_attributes(params[:album])
      if current_user.is_admin
      redirect_to edit_administration_path(@album.user.id)
      else
      redirect_to albums_url
      end
    else
      render :action => "edit"
    end
  end

  def destroy
    @album = Album.find(params[:id])
    @album.destroy
    if current_user.is_admin
      redirect_to ('/administration')
    else
      redirect_to ('/albums')
     end
  end

end
```

Fonction d'index des albums, elle cherche tous les albums qui ont été crées (pour un utilisateur donné en fait)

Fonction de création d'un nouvel album

Fonction pour aller dans l'édition d'un album, l'endroit où se trouve la possibilité de rajouter et supprimer des photos.

Fonction pour aller dans le menu d'édition du nom d'un album

Fonction pour la création d'un album

Fonction pour l'édition des attributs d'un album (par exemple son nom)

Fonction pour la destruction d'un album

Ecriture des modèles Album et Photo :
A ce stade il n'existait que ce qui concerne les photos et les albums, ainsi que les validations du modèle et non les autres conditions.

```
class Album < ActiveRecord::Base
  belongs_to :user
  has_many :photos, :dependent => :destroy
  has_one :coverflow, :dependent => :destroy
  validates_uniqueness_of :name, :scope => [:user_id]
  validates_presence_of :name
  validates_length_of       :name,     :maximum => 15
end
```

Nous voyons dans le modèle ci-dessus qu'un album appartient à un utilisateur (rajouté plus tard). Nous voyons également qu'un album a plusieurs photos et que les deux sont dépendants, autrement dit lors de la suppression d'un album les photos de l'album seront elles aussi supprimées. (idem pour le coverflow)
Enfin nous rajoutons des conditions sur les attributs d'un album, par exemple qu'un nom d'album doit être unique pour un utilisateur donné (:scope), qu'un album doit avoir un nom (i.e. pas blanc) et que ce nom ne doit pas dépasser 15 caractères.

Voici ensuite ce qui concerne le modèle Photo qui reprend les mêmes notions :

```
class Photo < ActiveRecord::Base

  def self.find_photos_posted
   find(:all)
  end

  belongs_to :album
  belongs_to :coverflow

  has_attachment :content_type => :image,
                 :storage => :file_system,
                 :max_size => 5.megabytes,
                 :thumbnails => {
                   :thumb  => '100x100>',
                    :cover  => '500x500>',
                 }

  validates_as_attachment
end
```

Has_attachment sert en revanche pour le plugin attachment fu, il précise le type de fichiers envoyés, la méthode de stockage dont nous avons déjà parlé, la taille maximum d'un fichier ainsi que les redimensionnements nécessaires faits sur les images pour créer des miniatures utiles pour différents affichages.

Installer ImageMagick et RMagick :

ImageMagick est un logiciel libre comprenant une bibliothèque ainsi qu'un ensemble d'utilitaires en ligne de commande permettant de créer, de convertir, de modifier et d'afficher des images dans un très grand nombre de formats, ainsi que de traiter ces images (redimensionnement, etc.). RMagick quant à lui est une interface Ruby pour ImageMagick. Ces deux logiciels nous sont nécessaires pour le redimensionnement de nos images afin de créer des miniatures utilisées dans les vues de notre application. Voici la procédure suivie :

Récupération de la dernière version de ImageMagick sur http://www.imagemagick.org.

```
tar xvzf ImageMagick.tar.gz
```

(Pour décompresser et désarchiver)

```
cd ImageMagick-X.Y.Z
```

(Pour se positionner dans le bon répertoire)

```
./configure --disable-static --with-modules --without-perl \
--without-magick-plus-plus --with-quantum-depth=8
```

(Pour configurer ImageMagick)

```
make
sudo make install
```

(Pour finir avec ImageMagick)

```
sudo gem install rmagick
```

(Pour installer RMagick à partir du RubyGems)

Renseigner la BDD :

Il convient ensuite de spécifier comment ces éléments seront décrits dans la base de données. Cela se fait par l'édition des fichiers schema.rb (qui donne un aperçu global de l'allure des tables) et des fichiers de migrations qui vont permettre de transférer ces informations vers la BDD.

Fichier schema.rb (parties relatives aux albums et photos)

```ruby
create_table "albums", :force => true do |t|
  t.string   "name"
  t.integer  "user_id"
  t.datetime "created_at"
  t.datetime "updated_at"
end

create_table "photos", :force => true do |t|
  t.string   "title"
  t.integer  "parent_id"
  t.integer  "size"
  t.integer  "width"
  t.integer  "height"
  t.string   "content_type"
  t.string   "filename"
  t.string   "thumbnail"
  t.integer  "album_id"
  t.datetime "created_at"
  t.datetime "updated_at"
end
```

Fichiers de migration

```ruby
class CreatePhotos < ActiveRecord::Migration
  def self.up
    create_table :photos do |t|
      t.string :title
      #Image upload related fields
      t.integer :parent_id, :size, :width, :height
      t.string :content_type, :filename, :thumbnail

      t.references :album

      t.timestamps
    end
  end

  def self.down
    drop_table :photos
  end
end
```

```ruby
class CreateAlbums < ActiveRecord::Migration
  def self.up
    create_table :albums do |t|
      t.string :name
     t.references :user
      t.timestamps
    end
  end

  def self.down
    drop_table :albums
  end
end
```

Ainsi en effectuant les commandes rake db :create (pour créer les bases de données) et rake db:migrate (pour effectuer la migration des tables), la base de données sous MySQL aura cette architecture désirée. L'opération de migration devra être effectuée après chaque changement des fichiers de migrations.
Mais pour cela il faut éditer correctement le fichier database.yml afin que Rails sache quel gestionnaire de base de données utiliser, et surtout comment y avoir l'accès.

database.yml

```
development:
  adapter: mysql
  encoding: utf8
  reconnect: false
  database: PhotosOnRails_development
  pool: 5
  username: root
  password: toor
  socket: /var/run/mysqld/mysqld.sock

test:
  adapter: mysql
  encoding: utf8
  reconnect: false
  database: PhotosOnRails_test
  pool: 5
  username: root
  password: toor
  socket: /var/run/mysqld/mysqld.sock

production:
  adapter: mysql
  encoding: utf8
  reconnect: false
  database: PhotosOnRails_production
  pool: 5
  username: root
  password:  toor
  socket: /var/run/mysqld/mysqld.sock
```

Nous remarquons ici qu'en fait notre application dispose de trois BDD, une utilisée pour la phase de développement, une pour la phase de production, et une pour la phase de test de l'application. Cette séparation est utile pour tout développeur qui souhaite effectuer ces tests ou modifications sans altérer le site fonctionnel.

Modifier la feuille de route :

Cette étape comme nous l'avons vu est très importante, sans cela l'application ne saura pas comment gérer correctement les requêtes.

Dans la feuille de route déjà présentée auparavant, seule une ligne concerne cette partie :

```
map.resources :albums, :has_many => :photos
```

Ecritures des vues :

Il n'y a ici rien de particulier à noter, la fonction des vues ayant déjà été expliqué c'est pourquoi nous n'avons pas jugé utile de décrire plus cette étape. A titre d'exemple, voici une partie de la vue permettant l'édition des photos d'un album (show.html.erb dans les vues de Album) :

```
<% if  @album.user==current_user || current_user.is_admin  %>

<% if @album.photos  %>

<div class="section">

    <% @album.photos.each do |photo| %>

    <div class="thumbnail">
        <%= lightbox_image_tag(photo.public_filename(:thumb), photo.public_filename(:cover) ,
        {:class=>"images"}, :title => photo.title, :rel => "lightbox[photos]") %>
        <div>
          <%= link_to 'Delete Photo', album_photo_url(@album, photo),
          :confirm => 'Are you sure?', :method => :delete %>
        </div>
    </div>

    <% end %>
</div>
<% end %>
```

etc...

Conclusion de cette étape :
A ce stade nous obtenons une application déjà fonctionnelle, avec les fonctions de bases. Pas d'utilisateurs, mais possibilité de créer des albums, d'y rajouter et supprimer des photos, de changer le nom d'un album ou de le détruire.

2.4.5 - Le rajout des autres fonctions

L'étape précédente étant relativement détaillée, nous nous contenterons désormais de citer les étapes dans l'ordre dans lequel elles ont été effectuées.

- Rajout du « MenuAlbums » permettant l'accès rapide à l'ensemble des albums dans le but futur d'y inclure un lien vers l'édition (ce qui a été crée précédemment) et un lien vers une vision des albums sous forme de Coverflow. Cela n'entraine aucune modification dans la BDD, simplement la création d'un contrôleur, d'un modèle et d'une vue.

- Rajout de la fonction Coverflow. Pour cela, création d'un fichier de migration et de modification de schema.rb. Création d'un contrôleur, d'un modèle et d'une vue.

- Rajout du système de login/utilisateurs. Pour cela, installation du plugin restful authentication

  ```
  script/plugin install git://github.com/technoweenie/restful-
  authentication.git
  ```

  Génération de l'authentification :
  ```
  script/generate authenticated user sessions
  ```

L'option –include-activation (pour permettre la confirmation des comptes par email) n'a à ce stade pas été choisi car elle entrainait d'autres modifications que nous n'étions pas sur d'avoir le temps de faire.

Modification de la feuille de route :

```
map.signup '/signup', :controller => 'users',   :action => 'new'
map.login  '/login',  :controller => 'session', :action => 'new'
map.logout '/logout', :controller => 'session', :action =>
'destroy'
```

Les vues, contrôleurs et modèles sont automatiquement générés par le plugin, seules quelques modifications légères ont été opérées : Déplacement des boutons login/logout sur le layout, modifications mineures des contrôleurs et modèles, mise en place dans le contrôleur de l'application de la ligne `include AuthenticatedSystem`. Celle-ci permet de rendre effectif plusieurs méthodes dans les autres contrôleurs qui héritent tous de cette classe, comme par exemple `login_required`. Comme son nom l'indique cette méthode permet de ne pas donner accès à toutes (ou certaines) méthodes d'un contrôleur sous la condition d'être un utilisateur authentifié.

- Création du panneau d'administration. Pour ceci il a d'abord fallu trouver ce qui définirait un administrateur. Le choix a été fait d'inclure une variable booléenne `is_admin` dans la table Users permettant la distinction d'un simple utilisateur d'un administrateur (et donc modification des fichiers de migrations et de schema.rb). Les contrôleurs déjà existant ont tous dû subir des modifications afin de créer des redirections pour l'administrateur permettant à ce dernier de parcourir le site de la manière décrite dans le schéma. A cet effet, de nouvelles vues et la modification d'anciennes ont été effectués. Il a également fallu créer des fonctions `update` et `destroy` dans users_controller.rb afin de permettre à l'administrateur (utilisateur) de pouvoir modifier ou détruire un (son) compte.

- Création d'une page « Erreur 404 » personnalisée. Etant donnée que celles-ci ne sont normalement effectives qu'une fois la phase de mise en production terminée et que nous n'étions pas sûr de pouvoir l'effectuer avant la rédaction du rapport, le choix a été fait de créer une méthode dans le contrôleur de l'application ainsi qu'un ajustement de la feuille de route afin de rendre cette page effective même en cas de présentation en phase de développement.

Méthode rescue_404

```
def rescue_404
render :file => "#{RAILS_ROOT}/public/404.html" ,
:status => "404 Not Found"
End
```

Rajout dans routes.rb

```
map.connect '*path',
:controller => 'application',
:action => 'rescue_404'
```

- Installation de Mongrel et Apache en vu d'une éventuelle mise en phase de production. Mongrel étant un serveur HTTP écrit en Ruby et en C conçu pour être léger et rapide. Apache (ou Apache HTTP Server) est un logiciel de serveur http, le plus populaire sur le web.

2.5 – Difficultés rencontrées

2.5.1 – De nouvelles notions

Le principal intérêt mais également la principale difficulté de ce TER fut l'acquisition de nouvelles connaissances dans lesquelles nous n'avions jusqu'alors aucune notion du fait de notre parcours antérieur essentiellement mathématique. Ainsi il nous a fallu apprendre à jongler entre ces différentes connaissances.
Du fait du nombre important de notions que requiert ce sujet il nous a fallu apprendre à se concentrer uniquement sur les points qui nous seraient utiles, chose qui n'est pas aisée du fait que nous ne savions pas toujours de quoi nous aurions besoin. Aussi une part importante de notre travail consista à la lecture de divers tutoriels et cours trouvés sur internet ou fournis par notre professeur référent (comme par exemple le livre Agile Web Development sur la pratique de Ruby On Rails). Un des aspects également frustrant lors de l'acquisition de ces connaissances fut de devoir parfois les appliquer sans forcément avoir eu le temps nécessaire pour les maitriser totalement ou d'avoir un temps de recul face à ce flot d'informations.
Ainsi du temps a également été perdu à cet effet. Non pas que ces heures de pratique soient une réelle perte de temps, car elles contribuent à l'élargissement de nos connaissances, mais il aurait parfois été préférable de pouvoir court-circuiter ces étapes au profit d'autres directement en rapport avec notre application. Ce fut par exemple le cas avec l'apprentissage des bases sur les BDD où nous nous sommes lancés dans la construction de bases de données « manuellement » à l'aide des commandes de MySQL, avant de découvrir par la suite que Rails permettait de le faire sans.

Néanmoins nous en avons toujours retiré des profits, et il nous semble que même si leur utilité directe pour l'application n'est pas avérée, cela nous aura permis de se poser parfois d'autres questions qui ont pu nous guider sur d'autres étapes.

2.5.2 – Des problèmes logiciels

Un autre aspect important de nos difficultés fut les différents problèmes auxquels nous nous sommes confrontés lors de l'utilisation ou l'installation des logiciels nécessaires au développement de notre application. Cela s'avéra vrai dès le départ avec l'installation de Ruby et Ruby On Rails. Tout le monde a déjà par le passé rencontré au moins une fois des problèmes logiciels sur sa machine avec les difficultés que cela entraine et la nécessité de rechercher des renseignements auprès de personnes plus spécialisées dans le domaine ou sur internet.

Il y eu également des problèmes de version de nos logiciels par rapport aux tutoriels que nous avons suivi pour nous familiariser avec RoR. En effet nous disposions de Rails 2 tandis qu'un grand nombre de nos tutoriels étaient conçus pour Rails 1. La principale différence étant dans la génération des scaffold dont nous n'avons pas parlé auparavant. En effet, la méthode du scaffolding consiste en une commande à générer une vue, un contrôleur et un modèle basique à modifier à notre gré ensuite. Cette méthode est donc relativement pratique et très utilisée au sein des petits tutoriels de familiarisation de RoR. Elle n'a ensuite pas été utilisée par nos soins car la vue générée était trop basique pour partir de celle-ci. Nous avons donc préféré générer séparément modèles et contrôleurs puis créer nos vues par la suite.
Pour information, Rails 3 est désormais disponible, illustrant l'essor et la progression rapide que connait Rails.

2.5.3 – La coordination du travail

Pour conclure sur cette partie relative aux difficultés, il était nécessaire d'aborder un des aspects important du projet, le travail en groupe. En effet, il n'est pas toujours aisé de se coordonner, de se répartir les tâches, d'autant plus dans un projet informatique où de nombreuses composantes sont dépendantes entre elles. L'aménagement de plages horaires communes a donc été fait afin de pouvoir se réunir, cela était essentiel du fait également que nous ne disposions que d'un seul ordinateur portable. Néanmoins il a malgré tout fallu devoir se réserver de nombreux temps de travail personnels en plus de ces quelques plages horaires communes.

2.6 – Graphisme du site

2.6.1 – Gimp

Gimp (pour GNU Image Manipulation Program) est un logiciel libre de traitement d'images souvent présenté comme une alternative au très connu Adobe Photoshop. Le format dédié de GIMP est le format XCF et permet de conserver les calques, canaux, et autres paramètres propres à une image éditée avec GIMP (l'équivalent du format PSD sous Adobe Photoshop). Il est néanmoins tout à fait possible d'ouvrir un fichier PSD sous Gimp. L'image conservera un grand nombre des propriétés du PSD permettant des retouches faciles, mais quelques pertes auront néanmoins lieu, comme la transformation d'une zone de texte en un calque d'image basique.

- **Gestion des couleurs**

GIMP possède une palette de sélection des couleurs d'arrière et premier plan ainsi qu'un outil « pipette » permettant de prélever des couleurs sur une image. Il utilise également les dégradés en les intégrant dans ses outils comme les brosses et l'outil de remplissage. Il possède par défaut une grande variété de dégradés de couleurs, et tout comme les brosses (formes de pinceau) il est possible d'en créer ou d'en télécharger de nouveaux (via des sites comme par exemple deviantArt). De nombreuses brosses existent également sous Photoshop et sont supportées par Gimp depuis la version 2.6 (les ressources y sont d'ailleurs plus grandes du fait d'une utilisation plus massive).

- **Les calques**

GIMP possède des outils de sélection variés comme la sélection rectangulaire, elliptique ou à main levée. Il peut aussi agir sur la taille et la position de la sélection sans en transformer le contenu, etc. Concernant sa gestion des calques, il permet de rendre des couches de l'image visibles, invisibles, ou transparentes.

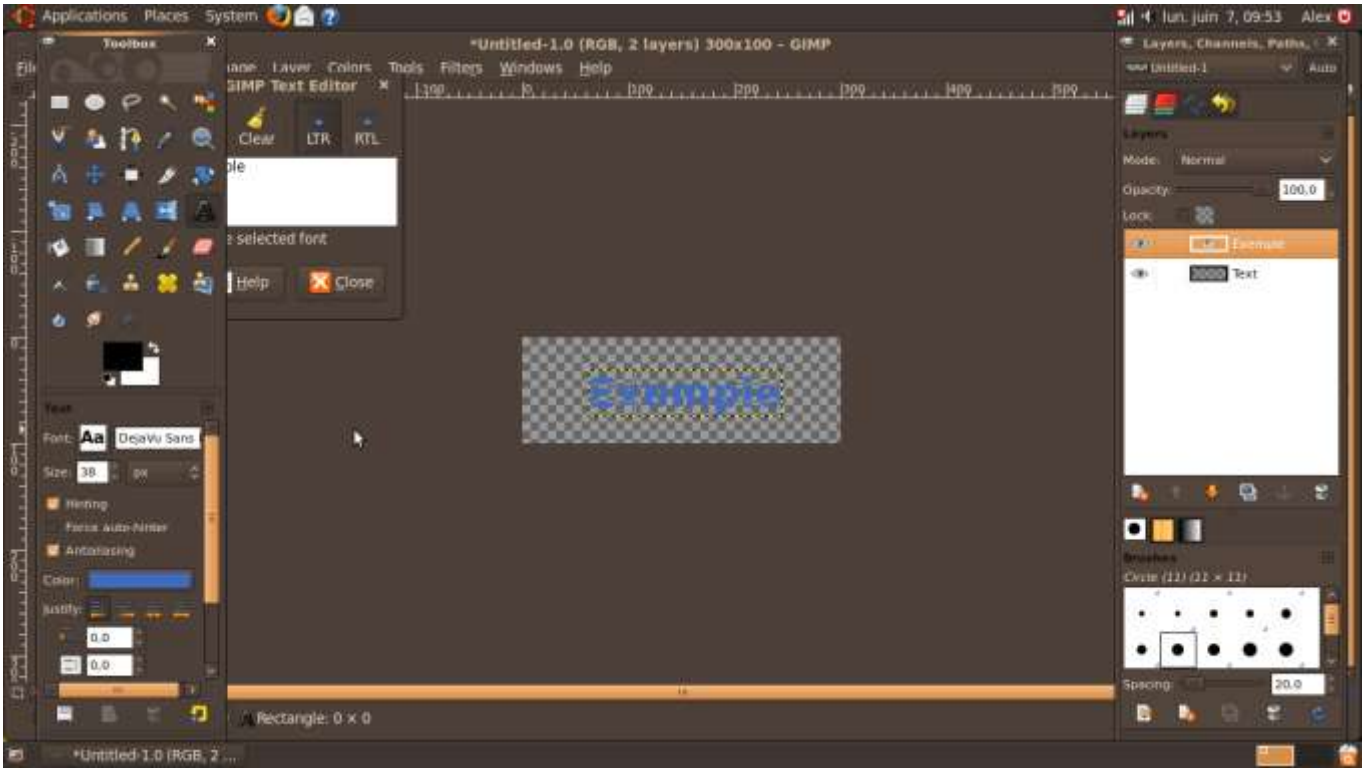
- **Effets, scripts et filtres**

GIMP possède par défaut à peu près 150 effets et filtres classés par types (flou, distorsion, artistique…). Il est également possible de créer ou télécharger des scripts, GIMP supportant comme langage : Perl, Tcl ou Python. Le support du langage Ruby n'est pour l'instant qu'au stade expérimental.

Exemple de l'utilisation d'un script :

Nous allons ici utiliser un des scripts de Gimp (Effet XCache) afin de donner un effet de relief à du texte. Nous partons à la base d'un unique calque vide sur lequel nous écrivons le texte désiré.

Interface Gimp



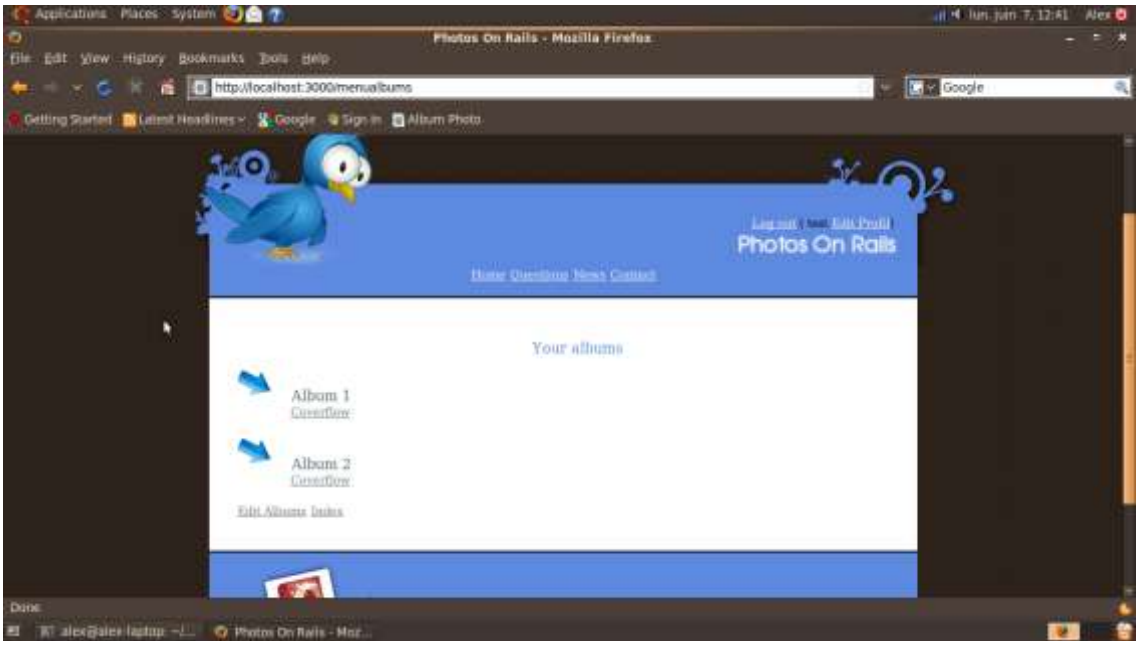
Vous remarquerez ici les différents menus de Gimp : à gauche les principaux outils et à droite les menus pour gérer canaux, calques, etc. Nous allons ensuite nous diriger dans le menu « Windows » pour y sélectionner le script désiré. En fait aucune manipulation supplémentaire n'est nécessaire, nous obtenons directement le rendu désiré. Il suffit de prendre soin d'enregistrer l'image au format PNG si le fond transparent veut être conservé.

2.6.2 – Vues d'ensemble

Voici quelques vues d'ensemble de notre application. Nous ne rentrerons pas dans les détails de la réalisation de son aspect. Seul le logo (oiseau bleu) n'a pas été crée par nos soins mais récupéré sur une recherche Google Images. Cette réalisation a nécessité à la fois un travail sur le CSS et sous Gimp. Le rendu final lors de la présentation est susceptible d'être modifié.

Menu des utilisateurs

Menu des Albums



Erreur 404 Personnalisée

3. Audit de l'application

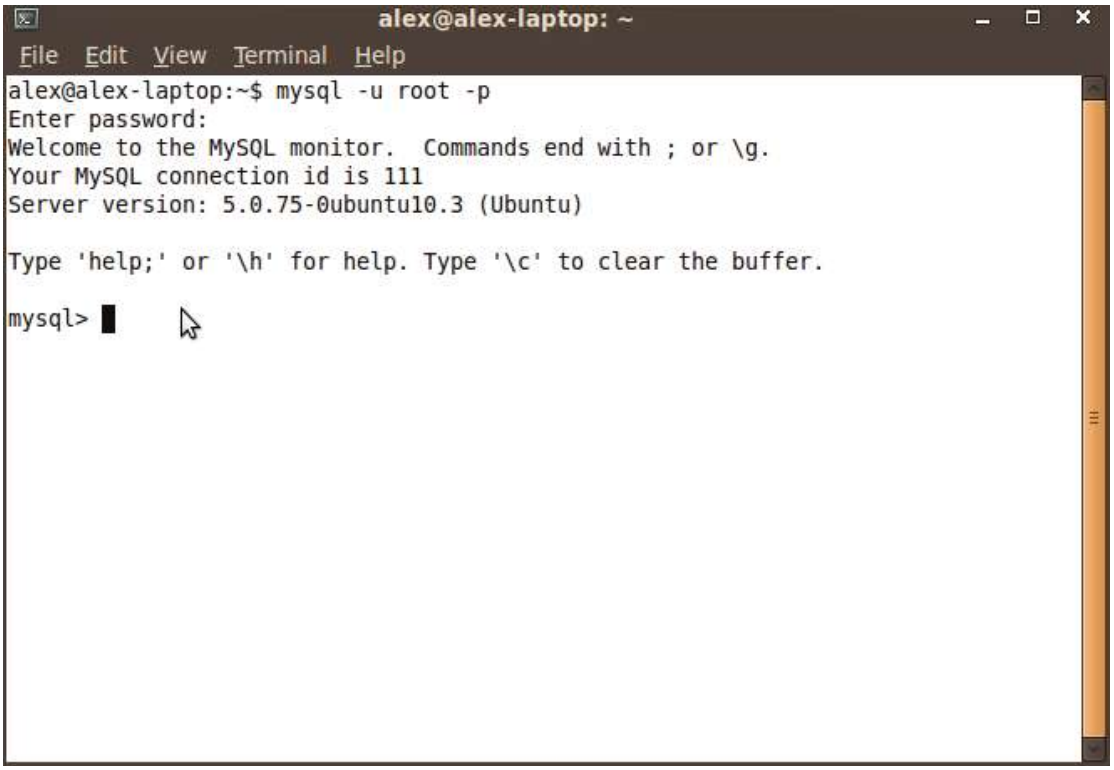
3.1 – Types de failles recherchées, solutions et outils appropriés

3.1.1 – Stockage des mots de passe

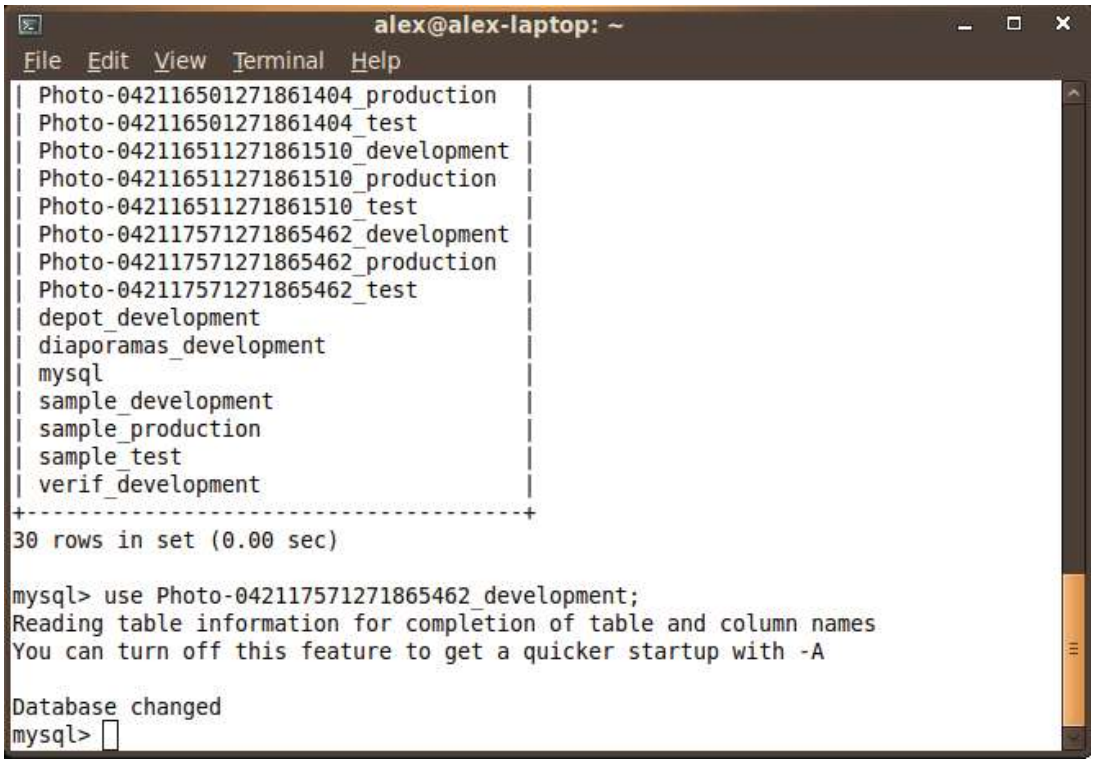
Le premier de nos soucis dans la sécurisation de notre application se situa au niveau du stockage des mots de passe des utilisateurs. C'est pourquoi notre choix s'est orienté vers un plugin d'authentification comme Restful (de nombreux autres existent), celui-ci proposant une solution simple et efficace. Nous avons déjà précédemment expliqué de quelle manière Restful permettait cette sécurisation par une méthode cryptage. Il ne restait alors qu'à vérifier que cette solution était bien effective en rentrant directement dans la BDD afin d'y regarder les mots de passe.

Contexte : Enregistrement sur le site d'un nouvel utilisateur (Test) avec pour mot de passe « azerty ».

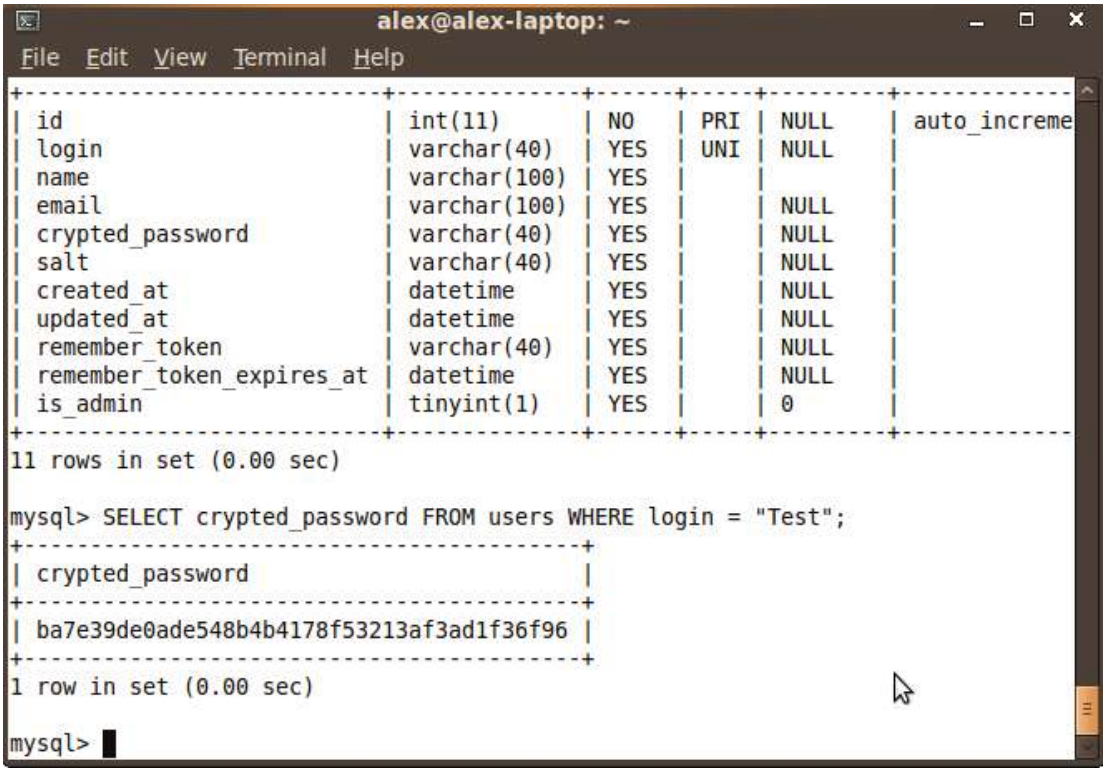
Connexion à MySQL



Sélection de la BDD
(Ici Photo-042117571271865462_development)



Visualisation des données



Ainsi nous pouvons nous rendre compte que le mot de passe de cet utilisateur n'est pas visible en clair dans la BDD. Ceci nous garantit qu'en cas de vol de notre BDD, les mots de passe ne pourront pas être récupérés.

3.1.2 – Jeu avec les URL

Comme nous l'avons vu, les URL de notre application nous renseignent pour beaucoup sur ce qui se passe en arrière plan. Cette affichage explicite des URL est utile lors de la navigation et pour que les utilisateurs se repèrent, mais peut aussi s'avérer être un point faible. En effet un utilisateur curieux pourrait éventuellement tester différentes URL en changeant seulement la partie relative aux id afin de tenter d'accéder aux albums ou profils des autres utilisateurs.
La solution à ce problème a en fait été appliquée en même temps que le développement de l'application. Il s'agissait alors d'écrire correctement nos vues notamment à l'aide de tests car celles-ci dépendent ainsi directement de l'utilisateur logué, et donc il ne peut pas apparaitre de données relatives à un autre utilisateur. D'autre part il s'agissait d'interdire via les contrôleurs ces mêmes vues dans le cas où l'utilisateur n'est pas logué. Ceci correspond à la fonction de la ligne `login_required` au début de nos contrôleurs.

3.1.3 – Transfert des données

Le stockage des mots de passe est certes une problématique importante, mais le transfert d'informations l'est encore plus. En effet les mots de passe ont beau être sécurisés, cette précaution s'avère inutile si les récupérer lors des transferts de données peut se faire facilement. N'oublions pas que la sécurité d'une application correspond à celle de sa composante la plus faible. Pour analyser cela l'outil Wireshark est tout à fait adapté.

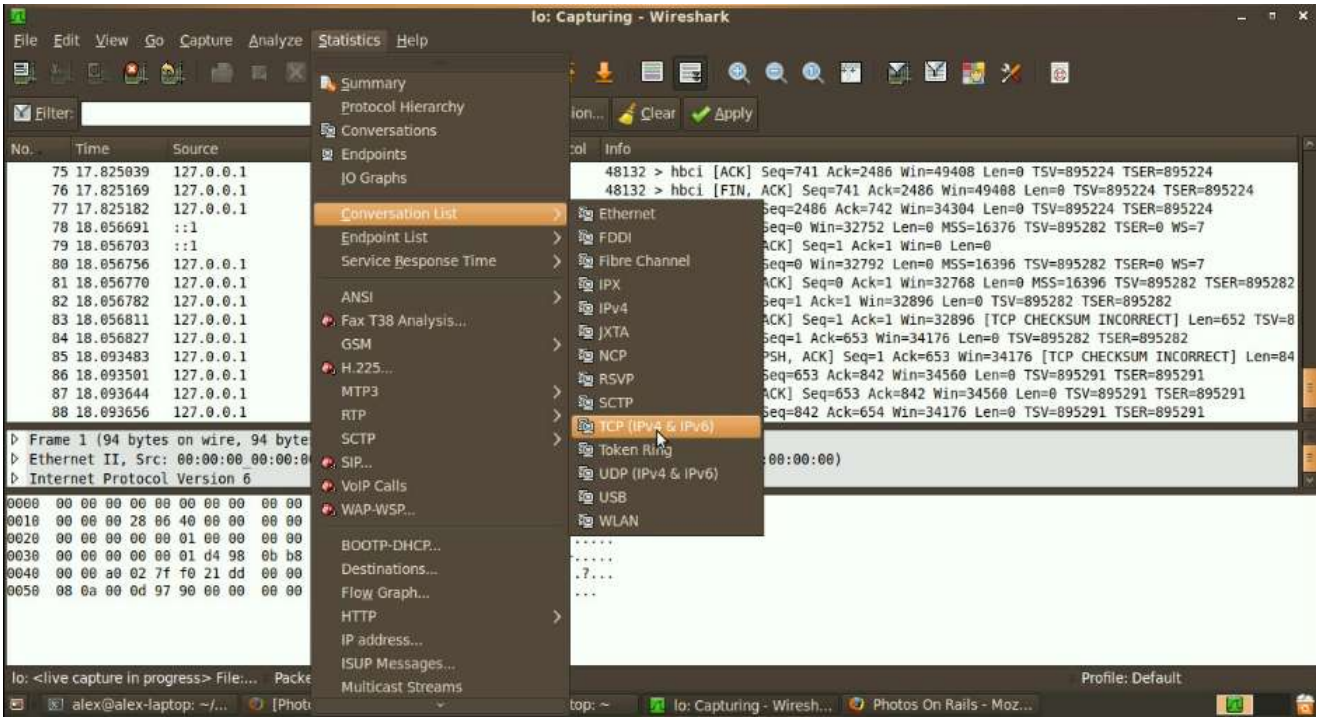
Wireshark est un analyseur de protocole qui examine les données à partir d'un réseau ou d'une capture de fichier. Il propose une navigation interactive sur les données capturées ainsi qu'un résumé pour chaque ensemble. Il possède entre autres des fonctions très utiles telles que l'affichage de langage filtré et la possibilité de visionner le flux reconstitué de la session TCP. C'est ce dernier point qui nous intéresse particulièrement.

Transmission Control Protocol (ou TCP) est un protocole de transport. L'application transmet des flux de données sur une connexion réseau, et TCP découpe le flux d'octets en segments à cet effet.

Contexte : Les tests sont effectués en phase de développement, à ce stade l'application est lancée sur un serveur en local sur le port 3000. Un utilisateur se rend sur la page d'accueil de notre site et décide de s'authentifier. Pendant ce temps Wireshark est lancé afin d'écouter les transferts de données entre l'application sur le serveur et l'utilisateur.

Sélection interface

Filtrage des données TCP



Reconstitution du flux

Voilà donc mis en évidence un des principaux soucis de notre application qui n'a pas pu être géré par manque de temps. Néanmoins en cas de mise en production de l'application, un moyen efficace de résoudre cela serait la mise en place d'un transfert de données sécurisé lors de l'authentification via SSL qui est un protocole de sécurisation des échanges sur Internet (les fameux https et cadenas qui apparaissent dans l'URL…)

3.2 – Autocritique de l'application

Nous finirons ce rapport avec une petite autocritique de notre application dans son ensemble. Du fait qu'il s'agisse pour nous d'une première pour un projet d'une telle ampleur, et que jamais auparavant nous n'avions eu à gérer tant de notions en même temps le rendu final de notre application n'est bien entendu pas exempt de tout défauts, notamment dans la manière de procéder qui aurait pu être différente.

3.2.1 – Ordonnancement du projet

Avec le recul nécessaire la principale modification au niveau de l'ordre des étapes effectué se situe dans l'implémentation du système d'authentification et d'administration. En effet cette étape est au final un des points central de notre application, or il s'agit de la dernière étape effectuée. Une implémentation dès le départ aurait permis une optimisation de cette interface afin de concevoir la suite de cette application autour de ces systèmes plutôt que l'inverse. La raison de cette mise en place à la toute fin est simple. Du fait du temps imparti pour la réalisation du projet et qu'au départ les objectifs étaient bien moins élevés, nous ne savions pas encore que cette fonctionnalité serait rajoutée. Notre projet dans sa globalité s'est construit autour de cette situation : parti d'un projet basique nous avons souhaité au fur et à mesure rajouter d'autres fonctionnalités pour le rendre plus abouti et plus proche de ce que pourrait être une utilisation à grande échelle.

3.2.2 – Agencement du MVC et de la BDD

A ce niveau là aussi d'autres solutions auraient pu être prises. Notamment au niveau de « MenuAlbums » et des Coverflows.

- **Les coverflows**

En effet par manque d'expérience les coverflows ont été intégrés dans la base de données avec un contrôleur et un modèle spécifique pour cela. Or aucune information vitale aux coverflows ne se situe au final dans la BDD. De même ce modèle et ce contrôleur ne comportent que des méthodes et informations qui auraient pu se placer dans Albums. Au final les vues correspondantes auraient pu alors être des vues supplémentaire de Albums. Ainsi cela aurait permis un petit allégement de la BDD avec une table en moins. Cela peu paraitre anecdotique, mais à grande échelle de petites informations supplémentaires dans la BDD peuvent se retrouver être de grandes quantités d'informations à stocker.

- **MenuAlbums**

Il en va de même pour cette partie qui s'avère au final simplement une vue, un modèle et un contrôleur très sommaires, sans informations dans la BDD, avec également uniquement des méthodes et informations qu'il aurait été possible de mettre dans le contrôleur et le modèle de Users. Cela n'aurait au final pas changé énormément en termes de ressources mais aurait rendu l'architecture du site plus intuitive.

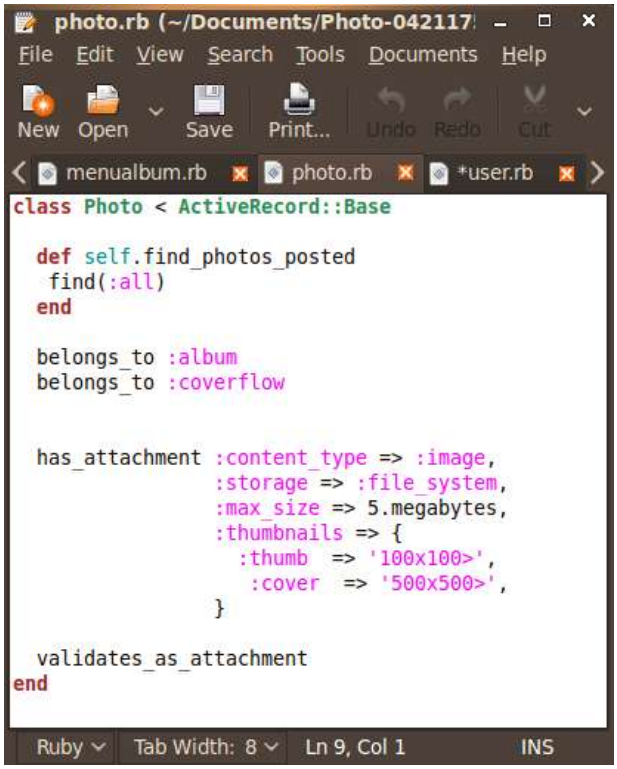
Conclusion

Pour conclure ce rapport, nous sommes désormais en mesure de dire que la réalisation de cette application nous aura permis d'acquérir de nouvelles connaissances dans des domaines informatiques très variés. Sans pour autant faire de nous de spécialiste dans ce secteur, cela aura été une introduction idéale pour le jour où nous aurons à fouiller plus en profondeur ces thématiques. Nous aurons également appris qu'il est important de bien définir ses buts quand le temps pour y parvenir est restreint. En effet nous avons certes rajouté des fonctionnalités qui ne nous étaient pas parues obligatoires au départ, mais il y aurait également eu possibilité de faire de multiples améliorations. Parmi celles-ci on pourrait retrouver le partage de photos entre les utilisateurs, une gestion de tags sur les photos, une méthode de classement des albums, possibilité de rendre certains albums publics, etc.

Fichiers annexes pour « Photos » :

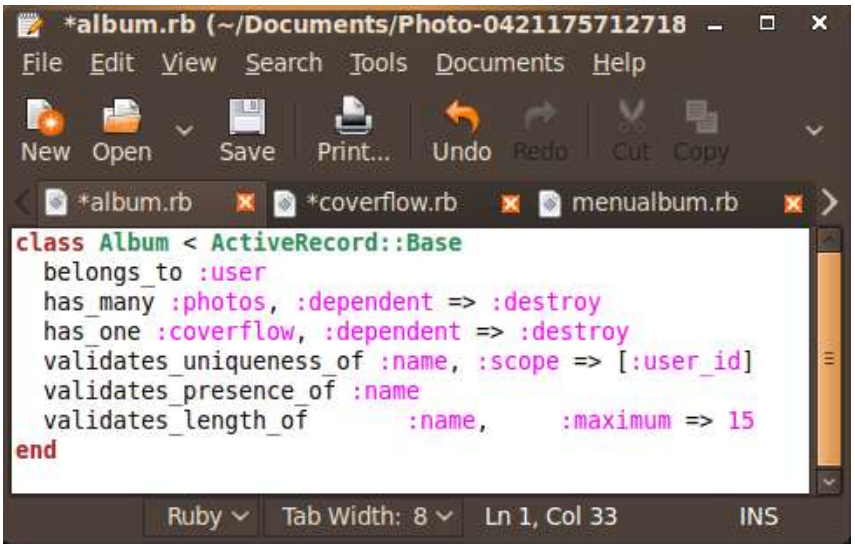
photos_controller.rb (Contrôleur)

photo.rb (Modèle)



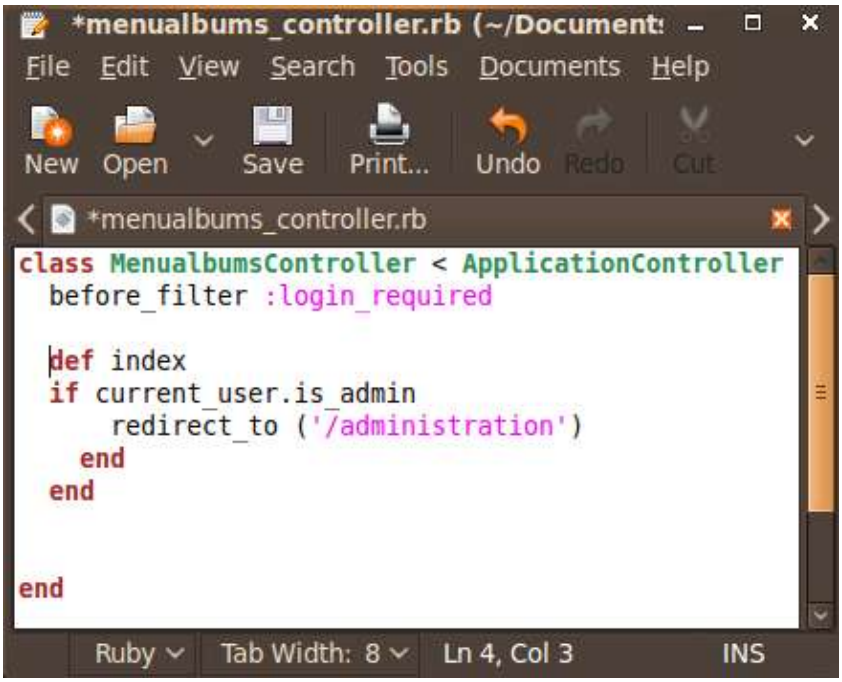
Fichiers annexes pour « Albums » :

album.rb (modèle)

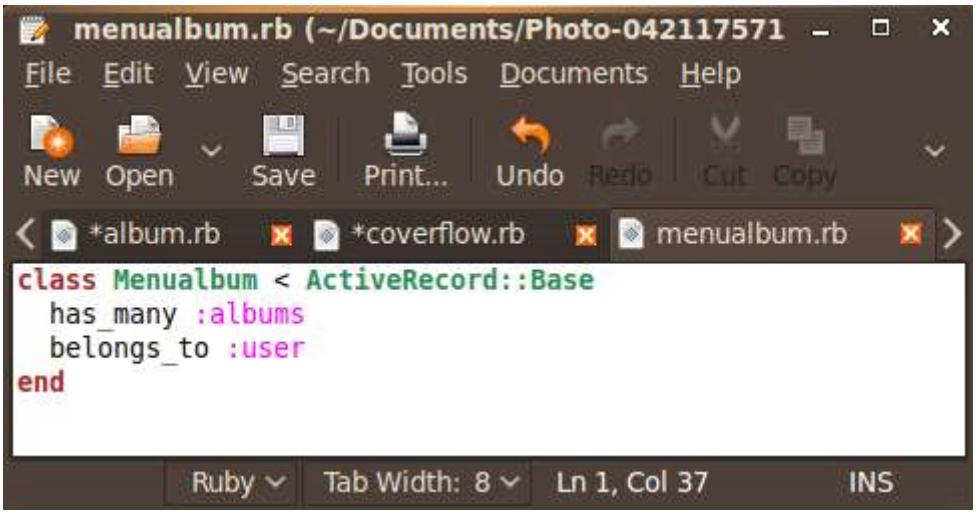


Fichiers annexes pour « MenuAlbums » :

menualbums_controller.rb (contrôleur)

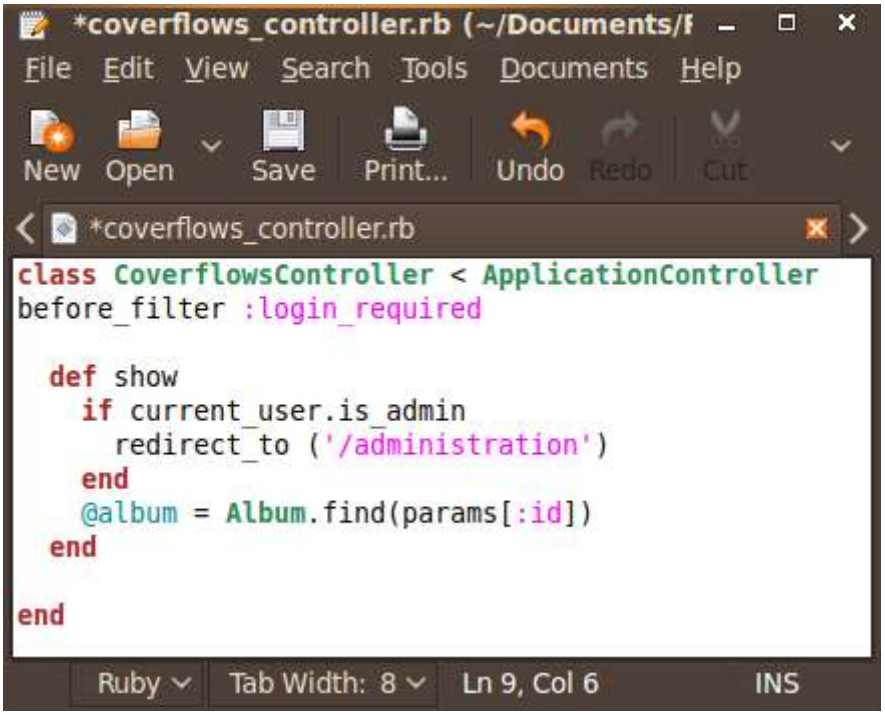


Menualbum.rb (modèle)

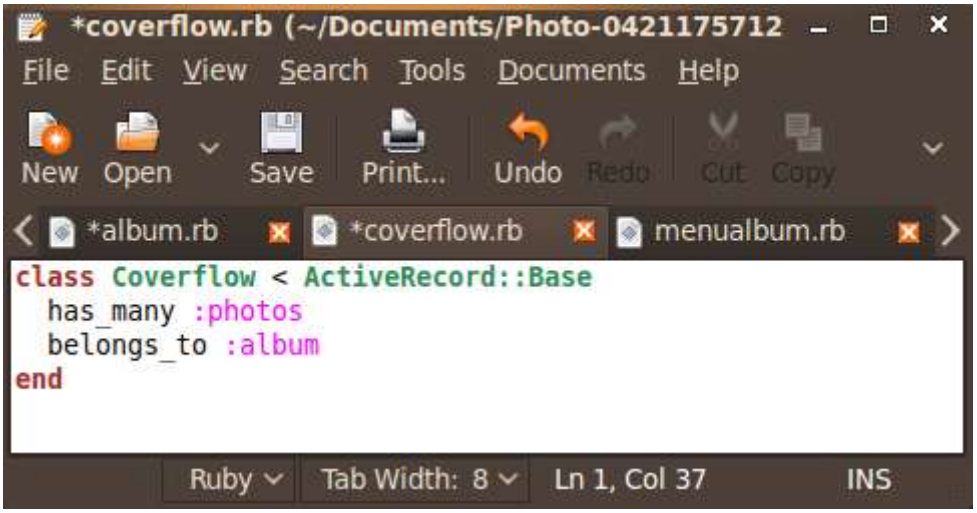


Fichiers annexes pour « Coverflows » :

coverflows_controller.rb (contrôleur)

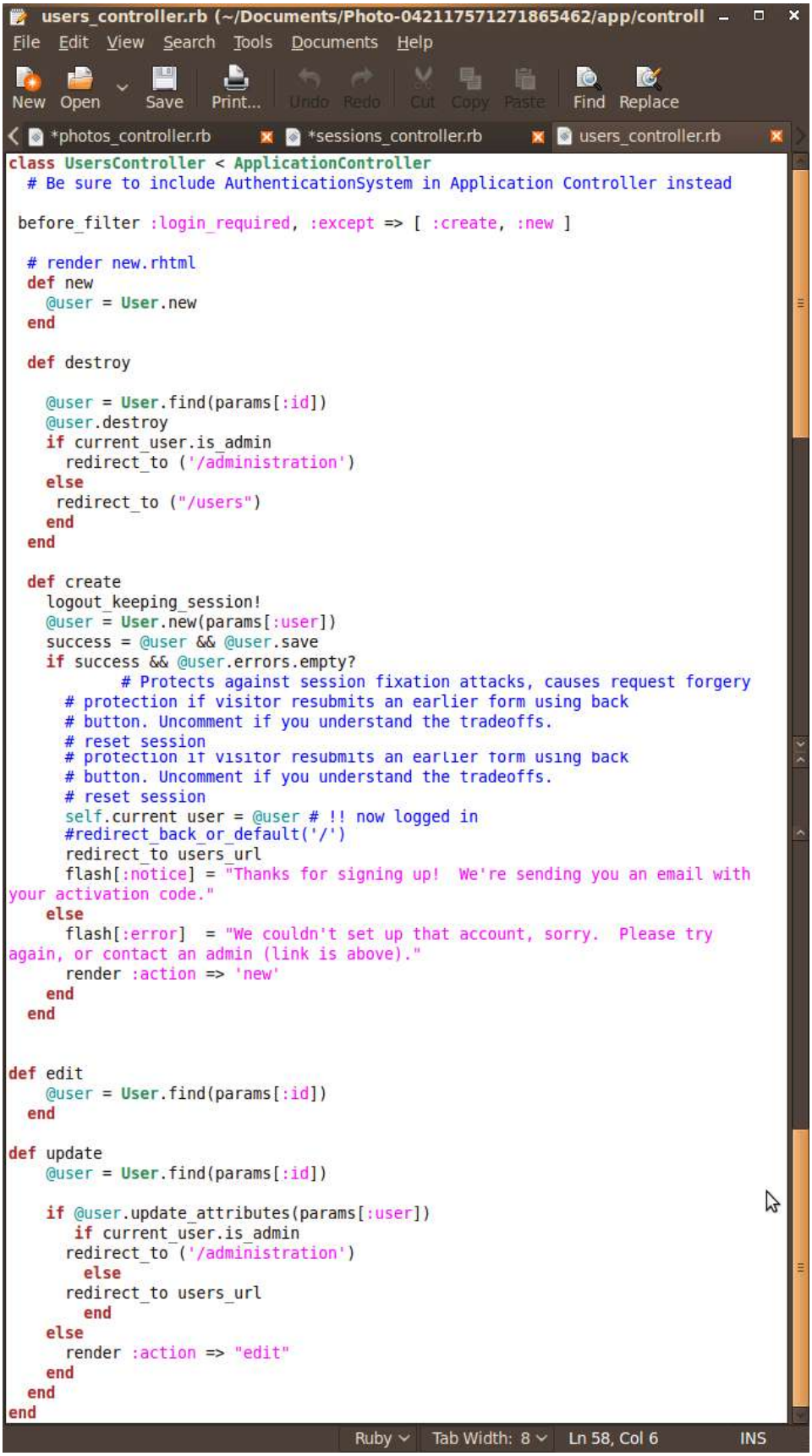


coverflow.rb (modèle)



Fichiers annexes pour « Users » :

users_controller.rb (contrôleur)



User.rb (modèle)

*user.rb (~/Documents/Photo-04211757127186546 2/app/models)

File Edit View Search Tools Documents Help

New Open Save Print... Undo Redo Cut Copy Paste Find Replace

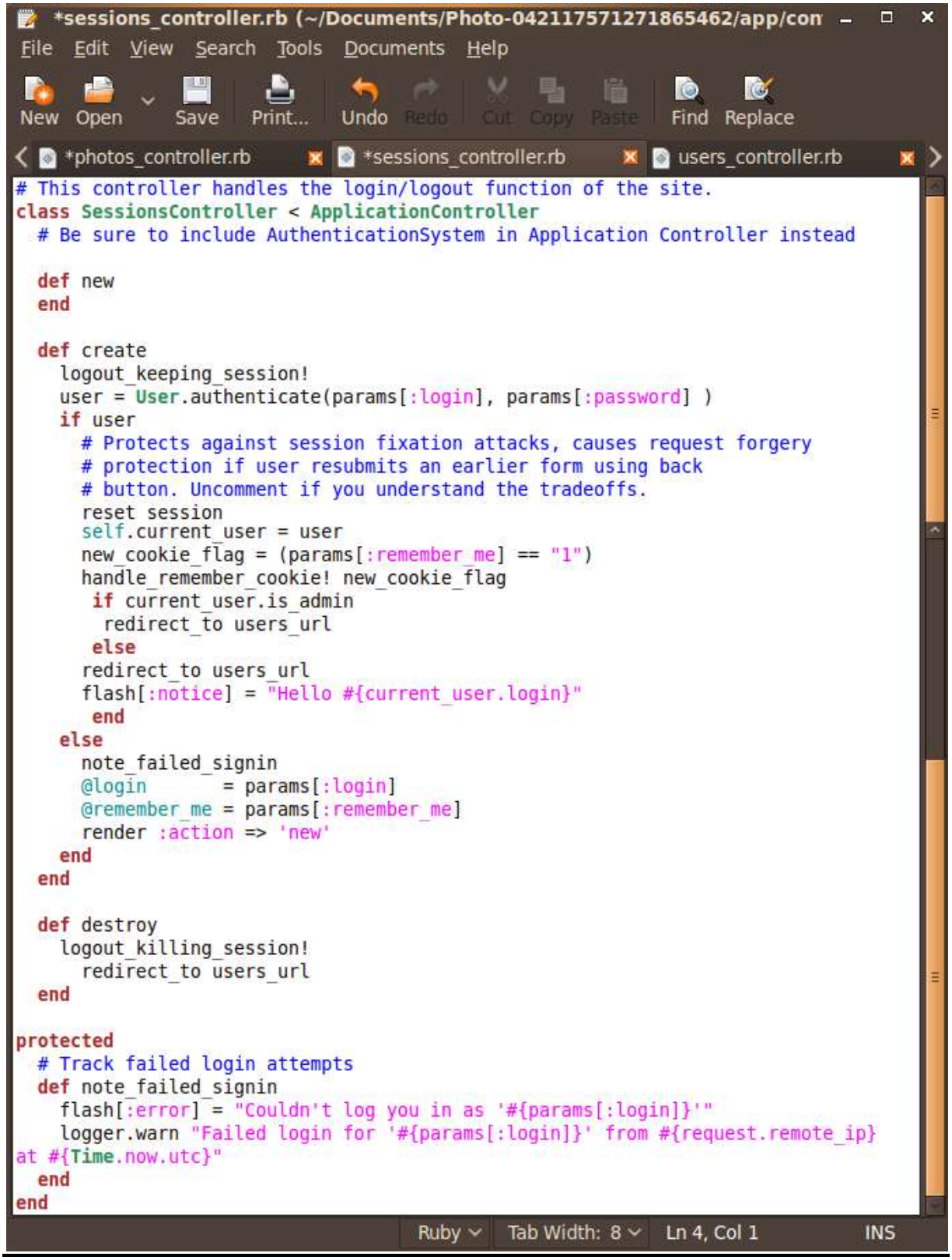
*coverflow.rb | menualbum.rb | photo.rb | *user.rb

```
require 'digest/sha1'

class User < ActiveRecord::Base
  include Authentication
  include Authentication::ByPassword
  include Authentication::ByCookieToken

        has_many :albums , :dependent => :destroy
  validates_presence_of     :login
  validates_length_of       :login,    :within => 3..40
  validates_uniqueness_of   :login
  validates_format_of       :login,    :with =>
Authentication.login_regex, :message => Authentication.bad_login_message

  validates_format_of       :name,     :with =>
Authentication.name_regex,  :message =>
Authentication.bad_name_message, :allow_nil => true
  validates_length_of       :name,     :maximum => 15

  validates_presence_of     :email
  validates_length_of       :email,    :within => 6..100 #r@a.wk
  validates_uniqueness_of   :email
  validates_format_of       :email,    :with =>
Authentication.email_regex, :message => Authentication.bad_email_message



  # HACK HACK HACK -- how to do attr_accessible from here?
  # prevents a user from submitting a crafted form that bypasses
activation
  # anything else you want your user to change should be added here.

attr_accessible :login, :email, :name, :password, :password_confirmation



  # Authenticates a user by their login name and unencrypted password.
Returns the user or nil.
  #
  # uff.  this is really an authorization, not authentication routine.
  # We really need a Dispatch Chain here or something.
  # This will also let us return a human error message.
  #
  def self.find_users
        find(:all)
  end

  #def albums_of_a_user
   # @user = User.find(params[:id])
    #@albums = @user.albums.find(params[:id])

  #end

  def self.authenticate(login, password)
    return nil if login.blank? || password.blank?
    u = find_by_login(login.downcase) # need to get the salt
    u && u.authenticated?(password) ? u : nil
  end

  def login=(value)
    write_attribute :login, (value ? value.downcase : nil)
  end

  def email=(value)
    write_attribute :email, (value ? value.downcase : nil)
  end

  protected

end
```

Ruby Tab Width: 8 Ln 63, Col 1 INS

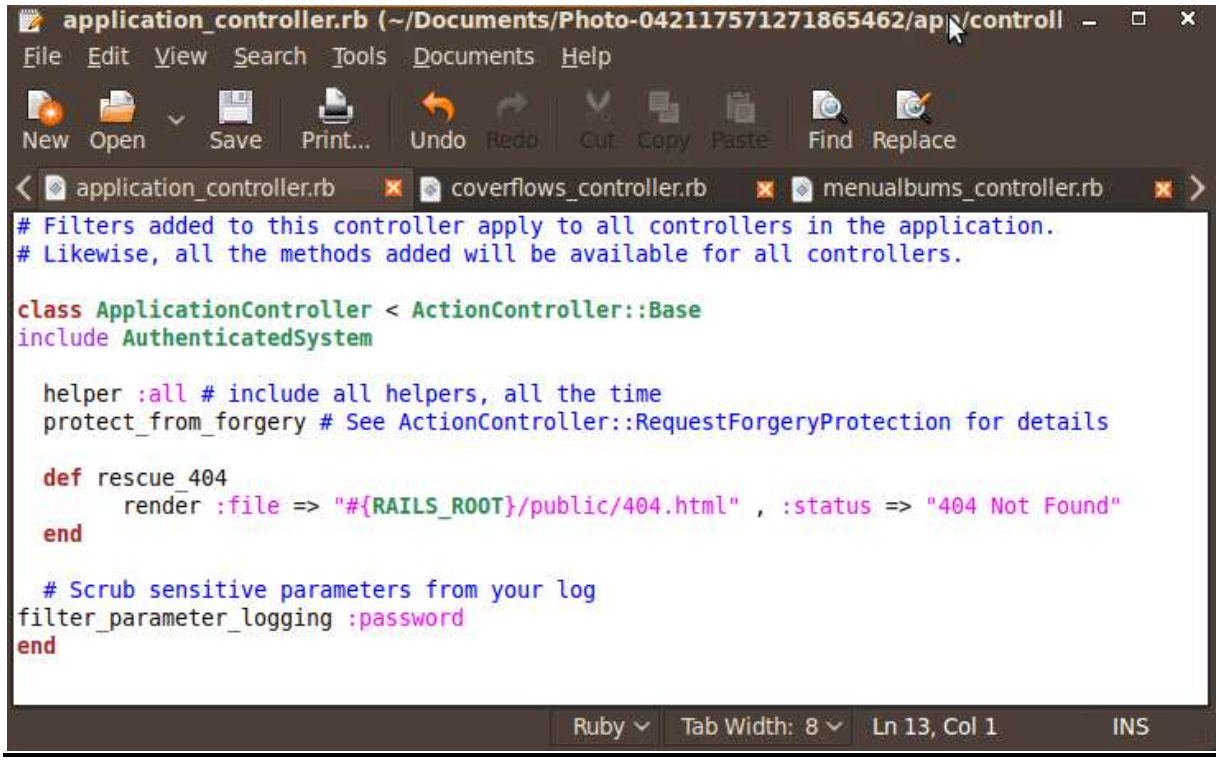
Fichiers annexes pour « Sessions » :

sessions_controller.rb (contrôleur)



Fichiers annexes pour l'application :

application_controller.rb (contrôleur)



Tutoriel Ruby On Rails :
(http://valentinbourgoin.net/blog/web/concevoir-une-application-ruby-on-rails-20-78/)

Utiliser Rails

Créer l'application

Pour créer une nouvelle application, placez-vous dans votre dossier personnel. Une simple commande vous permettra de générer toutes l'arborescence du futur projet, que nous nommerons **myMovies**.

```
rails myMovies
```

Lancer le serveur

Si vous travaillez en local, vous devrez lancer le serveur de développement intégré à RoR, WEBrick, pour que votre application soit accessible. Pour ce faire, tapez simplement :

```
ruby script/server
```

Une fois le serveur lancé, votre site sera accessible à l'adresse :

```
http://localhost:3000
```

Si vous travaillez sur un serveur distant, vous pourrez normalement visualiser directement votre application sans passer par le lancement manuel de WEBrick.

Créer et configurer la base

Par le biais de votre SGBD, commencez par **créer une base de données** pour votre projet, nommée - par exemple - myMovies.

Configurons maintenant l'accès à cette base, en éditant le fichier **/myMovies/config/database.yml** en renseignant le moteur (ici mySQL) l'hôte, l'utilisateur mySQL et le mot de passe. Note : il est possible de différencier les bases de développement et de production.

Une fois l'application configurée, on pourra simplement créer la base grâce à la commande :

```
rake db:create:all
```

Créer un modèle

Cette étape est cruciale car elle détermine toute **l'architecture** des données de notre application. Vous devrez déterminer toutes les entités qu'il vous sera nécessaire de stocker. D'une mauvaise conception découlerait une application bancale. Dans notre cas, la question ne se pose pas car nous avons choisi un cas de figure particulièrement simple. Nous n'avons à travailler que sur un seul modèle : **movie**. Bien sûr, il sera possible d'ajouter des entités plus tard (des commentaires, par exemple), mais faisons simple pour commencer.

Maintenant, réfléchissons sur le **données** que nous avons besoin de stocker pour chaque modèle : le titre du film, son réalisateur, l'année de sortie, une brève description et une url permettant d'en savoir plus (un lien imdb par exemple). Pour cela, lançons la commande :

```
ruby script/generate scaffold Movie title:string director:string year:string decr
```

Nous lançons ainsi le scaffold qui va générer tout le code lié à ce modèle, de la table dans la base de donnée aux vues génériques destinées à son affichage. Nous avons juste à mettre à jour la base de données :

```
rake db:migrate
```

L'interface générée

Voyons maintenant ce que nous avons généré avec cette simple commande. Après avoir lancé le serveur, allez sur

```
http://localhost:3000/movies
```

Nous avons maintenant une veritable interface **CRUD** pour l'ajout et l'édition de films. Essayez d'ajouter un nouveau film ; il apparaitra dans la liste et vous pourrez l'éditez ou le supprimer.

Validation du modèle

On peut aller plus loin dans la création du modèle. Si certaines des données sont obligatoire au bon fonctionnement de l'application, comment empêcher la saisie d'entrées incorrectes et mal formées ? C'est là qu'intervient la validation du modèle. Pour cela, ouvrez le modèle proprement dit, stocké dans le fichier **/myMovies/app/models/movies.rb**. Pour l'instant, le fichier est quasiment vide, et c'est normal. C'est ici que nous allons renseigner les informations relatives à la validation. Au sein de la classe, tapez (attention à la place des espaces) :

```
validates_presence of :title, :director, :year
```

C'est tout ! Dans l'application, essayez de saisir un film sans réalisateur ou sans date ; l'enregistrement échouera. Finis, les tests bancaux : tout est automatisé grâce RoR !

Personnaliser l'application

Comme vous pouvez le voir, sans que nous n'ayons besoin d'écrire la moindre véritable ligne de code, RoR nous a généré une interface complète CRUD (Create, Read, Update, Delete) pour notre modèle. On peut aisément ajouter des nouveaux films, éditer ou supprimer les existants, et visualiser le tout en détails. Libre à nous de personnaliser les vues grâce au système de **templates** que nous allons voir maintenant.
Vous l'avez vu, l'habillage de l'interface est très sommaire. A nous de l'adapter et de l'intégrer dans un design complet. Pour cela, rien de compliqué ; il ne s'agit que de XHTML / CSS standard. Regardez dans le dossier **/myMovies/app/views/movie/** : c'est ici que sont cachés les templates correspondant aux vues pour notre modèle. On y retrouve les différentes actions possibles : **l'affichage détaillé** (show.html.erb), **l'édition** (edit.html.erb), **l'ajout** (new.html.erb) et enfin la liste en **page d'accueil** (index.html.erb).

Ces templates sont incorporés dans une mise en page globale, définie elle par le ficher **myMovies/app /views/layouts/movies.html.erb** par le biais de la ligne <%= yield>. Vous pouvez ainsi modifier vos différentes vues, de la structure HTML au CSS (stocké dans /public/stylesheets) en mettant les mains dans le cambouis un minimum...